Portable Tools for Interoperable Grids

Modular Architectures and Software for
Job and Workflow Management

Johan Tordsson

PhD Thesis, March 2009
Department of Computing Science
Umeå University
Sweden

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

tordsson@cs.umu.se

ISBN 978-91-7264-754-1
ISSN 0348-0542
UMINF 09.08

Printed by Print & Media, Umeå University, 2009

Abstract

The emergence of Grid computing infrastructures enables researchers to share resources and collaborate in more efficient ways than before, despite belonging to different organizations and being geographically distributed. While the Grid computing paradigm offers new opportunities, it also gives rise to new difficulties. This thesis investigates methods, architectures, and algorithms for a range of topics in the area of Grid resource management. One studied topic is how to automate and improve resource selection, despite heterogeneity in Grid hardware, software, availability, ownership, and usage policies. Algorithmical difficulties for this are, e.g., characterization of jobs and resources, prediction of resource performance, and data placement considerations. Investigated Quality of Service aspects of resource selection include how to guarantee job start and/or completion times as well as how to synchronize multiple resources for coordinated use through coallocation. Another explored research topic is architectural considerations for frameworks that simplify and automate submission, monitoring, and fault handling for large amounts of jobs. This thesis also investigates suitable Grid interaction patterns for scientific workflows, studies programming models that enable data parallelism for such workflows, as well as analyzes how workflow composition tools should be designed to increase flexibility and expressiveness.

We today have the somewhat paradoxical situation where Grids, originally aimed to federate resources and overcome interoperability problems between different computing platforms, themselves struggle with interoperability problems caused by the wide range of interfaces, protocols, and data formats that are used in different environments. This thesis demonstrates how proof-of-concept software tools for Grid resource management can, by using (proposed) standard formats and protocols as well as leveraging state-of-the-art principles from service-oriented architectures, be made independent of current Grid infrastructures. Further interoperability contributions include an in-depth study that surveys issues related to the use of Grid resources in scientific workflows. This study improves our understanding of interoperability among scientific workflow systems by viewing this topic from three different perspectives: model of computation, workflow language, and execution environment.

A final contribution in this thesis is the investigation of how the design of Grid middleware tools can adopt principles and concepts from software engineering in order to improve, e.g., adaptability and interoperability.

Sammanfattning

Dagens Grid-infrastruktur gör det möjligt för forskare att dela vetenskaplig utrustning såsom högpresterande datorer och dyrbara instrument och därmed samarbeta mer effektivt än tidigare, trots att de tillhör olika organisationer och är geografiskt åtskilda. Grid-tekniken erbjuder nya möjligheter, men ger även upphov till nya problemställningar. Denna avhandling studerar en rad frågeställningar inom Grid, med särskilt fokus på metoder, arkitekturer och algoritmer för resurshantering. Ett bidrag är en studie av hur matchning av jobb och resurser kan automatiseras och förbättras, trots heterogenitet i hårdvara, programvaror och användningspolicyer i de maskiner som finns tillgängliga i en Grid. Algoritmiska aspekter av detta problem inkluderar karaktärisering av jobb och maskiner, prestandaprediktion samt konsekvenser av placeringen av in- och utdatafiler. Vidare studeras kvalitetsgarantier (Quality of Service), i detta fall vilka mekanismer som krävs för att garantera start- och/eller sluttider för jobb. Ett relaterat problem är hur man bäst samallokerar flera resurser för koordinerad användning. Ett annat bidrag är en studie av lämpliga arkitekturer för automatiserad hantering av stora mängder jobb. Avhandlingen behandlar även hur man på bästa sätt integrerar Grid-resurser i vetenskapliga arbetsflöden (workflows), vilka programmeringsmodeller som lämper sig bäst för dataparallellism för workflows, samt hur verktyg för att definiera workflows bör konstrueras för ökad flexiblitet.

Vi har idag en något paradoxal situation där Grid-teknik, som delvis designats med målet att integrera heterogena plattformar och överbrygga kompatibilitetsproblem, i sig ger upphov till en ny nivå av kompabilitetsproblem, mellan olika Grid-plattformar. Dessa problem beror på stora skillnader i såväl de gränssnitt, protokoll och dataformat som används i dagens infrastrukturer som i deras övergripande arkitektur. Denna avhandling demonstrerar hur programvara för resurshantering kan göras oberoende av nuvarande Grid-plattformar genom att utnyttja (föreslagna) standardformat och protokoll såväl som principer från service-orienterade arkitekturer. Andra bidrag inkluderar en fördjupad studie om hur Grid-resurser bäst integreras i workflows. Denna studie analyserar skillnader mellan befintliga workflow-system och belyser interoperabilitet mellan dessa ur tre aspekter: beräkningsmodell, språk för att beskriva workflows samt exekveringsplattform.

Avslutningsvis studeras i denna avhandling hur resultat av forskning inom programvaruteknik (software engineering) kan användas för att förbättra designen av Grid-programvaror, bland annat för att öka interoperabilitet och anpassningsbarhet.

Preface

This thesis consists of an introduction and the following papers:

Paper I E. Elmroth and J. Tordsson. Grid Resource Brokering Algorithms Enabling Advance Reservations and Resource Selection Based on Performance Prediction. *Future Generation Computer Systems. The International Journal of Grid Computing: Theory, Methods and Applications*, Elsevier B.V., Vol 24, No. 6, pp. 585-593, 2008.

Paper II E. Elmroth and J. Tordsson. An Interoperable, Standards-based Grid Resource Broker and Job Submission Service. In H. Stockinger, R. Buyya, and R. Perrot (Eds.), *e-Science 2005. First IEEE Conference on e-Science and Grid Computing*, IEEE Computer Society Press, pp. 212-220, 2005.

Paper III 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*, accepted, 2009.

Paper IV 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 (Eds.), *Towards Next Generation Grids*, Springer-Verlag, pp. 175 - 184, 2007.

Paper V E. Elmroth, F. Hernández, and J. Tordsson. A Light-weight Grid Workflow Execution Service Enabling Client and Middleware Independence. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewsky (Eds.), *Parallel Processing and Applied Mathematics*, Lecture Notes in Computer Science vol. 4967, Springer-Verlag, pp. 754 - 761, 2008.

Paper VI A-C. Berglund, E. Elmroth, F. Hernández, B. Sandman, and J. Tordsson. Combining Local and Grid Resources in Scientific Workflows (for Bioinformatics). In *9th International Workshop on State-of-the-Art in Scientific and Parallel Computing*, Lecture Notes in Computer Science, Springer-Verlag, accepted, 2009.

Paper VII E. Elmroth, F. Hernández, and J. Tordsson. Three Fundamental Dimensions of Scientific Workflow Interoperability: Model of Computation,

Language, and Execution Environment. Technical report UMINF 09.05. Submitted for journal publication, 2009.

Paper VIII E. Elmroth, F. Hernández, J. Tordsson, and P-O. Östberg. Designing Service-based Resource Management Tools for a Healthy Grid Ecosystem. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewsky (Eds.), *Parallel Processing and Applied Mathematics*, Lecture Notes in Computer Science vol. 4967, Springer-Verlag, pp. 259 - 270, 2008.

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 contracts 343-2003-953 and 621-2005-3667, as well as by the European Community's Seventh Framework Programme ([FP7/2001-2013]) under grant agreement no. 215605.

Acknowledgments

First of all, I thank my supervisor Erik Elmroth for guiding, encouraging, and most of all inspiring this work. Your day-round rapid email response times however worry me :-). I also thank my associate supervisor Bo Kågström for enlightening discussions. A big *gracias* goes to Francisco Hernández, with whom I have had the pleasure to work closely together with for the last two years. I express my gratitude to my fellow Grid researchers Peter Gardfjäll, Arvid Norberg, and P-O Östberg. I also acknowledge Ann-Charlotte Berglund Sonnhammar and Björn Sandman for their contributions to the work in this thesis.

Åke Sandgren, Björn Torkelsson and Tomas Ögren provided valuable assistance in the never-ending task of (re)installing our local Grid testbed. I am grateful to Inger Sandgren, who numerous times patiently helped me navigate the maze of travel expenses reimbursement. I also thank the Grid computing group and all other collegues for providing a creative research environment.

On a more personal level, I thank my friends and family for their encouragement.

Thank you!

Umeå, March 2009
Johan Tordsson

Contents

Chapter 1

Introduction

1.1 Background

During the late 1980's and early 1990's, the rapid development of network capacity made it feasible to interconnect remotely located parallel computers, in order to tackle larger-than-supercomputer problems. The resulting infrastructures were referred to as meta-computers. The earliest meta-computing platforms were constructed with experiment-specific protocols. The I-WAY project [78] developed the first general-purpose toolkit and was successfully used by more than 50 applications, demonstrating the feasibility of interconnecting resources such as supercomputers, instruments and storage. The term Grid computing was coined to describe the interconnection of more general types of resources. The choice of name reflects the vision of computing made available as a utility, in analogy with how electricity is available from the power grid. In addition to the possibility to tackle larger-than-supercomputer problems, early motivating factors for the construction of Grids were improved collaboration and the possibility to remotely access scarce and expensive scientific instruments.

Perhaps the most well known Grid middleware is the Globus Toolkit [90], a further development of the I-WAY software, with greater focus on protocols for fundamental tasks such as resource discovery, job submission and data transfer. Later versions of the Globus Toolkit are used as building blocks in many of todays Grids. Another early general-purpose Grid toolkit is Legion [98] that models and controls remote resources in an object-oriented manner. Further examples of early Grid projects include the Storage Resource Broker [22] that enables uniform access to heterogeneous storage resources; AppLeS [35] and NetSolve [34], both application-level schedulers; and Cactus [11], a programming environment for parallel high performance computing on various platforms, including Grids. An in-depth description of early Grid computing projects is beyond the scope of this thesis, and can be found in overview books [77, 27].

1.2 Characteristics of Grid Computing

Early Grid computing efforts focused on uniform, efficient and secure access to computational resources, despite heterogeneity in ownership, security mechanisms, and policies. An early definition states that Grid computing is "Coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations" [82]. In Grid computing, the term Virtual Organization (VO) is used to describe the parties sharing resources. The formation of a VO is motivated by a scenario where "a number of mutually distrustful participants with varying degrees of prior knowledge (perhaps none at all) want to share resources in order to perform some task" [82].

One commonly used definition is the three point check list [74] by Foster that defines a Grid as a system that:

1. coordinates resources that are not subject to centralized control,
2. uses standard, open, general-purpose protocols and interfaces,
3. delivers nontrivial Qualities of Service.

A more recent definition is given in the Open Grid Services Architecture (OGSA) glossary of terms by the Open Grid Forum (OGF). Here, a Grid is defined as "A system that is concerned with the integration, virtualization, and management of services and resources in a distributed, heterogeneous environment that supports collections of users and resources (virtual organizations) across traditional administrative and organizational domains (real organizations)" [59]. Although this definition is in large in harmony with the one by Foster, a few things are worth noting. First, there is a clear focus on exposing the capabilities of the Grid as services. This is a result of the recent trend of usage of service-oriented architectures in Grid computing, a topic further discussed in Section 6.1. The use of virtualization in the OGF definition should not be confused with virtualization by use of virtual machines (further explored in Section 8.2), but is rather the process of hiding differences in properties and operations of a set of similar resources and making these available for view and/or manipulation through a set of common interfaces [59].

Another recent Grid definition comes from the CoreGRID network of excellence that describes a Grid as a "fully distributed, dynamically reconfigurable, scalable and autonomous infrastructure to provide location independent, pervasive, reliable, secure and efficient access to a coordinated set of services encapsulating and virtualizing resources (computing power, storage, instruments, data, etc.) in order to generate knowledge" [47]. In the CoreGRID definition, the quality of service requirements (e.g., scalability, security, reliability) are made more explicit. Furthermore, the goal of the architecture has a clear focus on generation of knowledge, clearly inspired by the emergence of the e-Science discipline.

John Taylor defines e-Science as follows: "e-Science is about global collaboration in key areas of science and the next generation of infrastructure that

will enable it" [199]. In e-Science, there is an inherent focus on scientific data. Large amounts of data are processed and analyzed in a collaborative fashion by distributed teams of scientists. This requires access to high-end computing resources, distributed data storage, and visualization equipment, as well as the software tools required by the scientists to cope with exponentially growing amounts of data. Grid computing can hence be seen as an enabling technology for e-Science [103].

The research presented in this thesis is well aligned with the above definitions of Grid computing. The Grid scenarios addressed stress that resource management tasks must be performed despite lack of global control. One main objective is to demonstrate how open, standard protocols and interfaces can be used to achieve interoperability between different Grids. The thesis also investigates various issues for Quality of Service (QoS), with particular focus on deadline constraints.

1.3 Fundamental Grid Capabilities

Many of the existing Grid infrastructures have fundamental differences in purpose, architecture, and type of resources that they interconnect. However, almost all Grids have: a model to describe resource capabilities, mechanisms to discover available resources and monitor known ones, resource allocation interfaces, data management functionality, and a security infrastructure. General discussions of Grid capabilities are found, e.g., in [84, 193]. The following descriptions focus on capabilities required by resource selection and job management systems. Most Grid infrastructures provide the discussed capabilities, although the used protocols, interfaces, and software components sometimes have fundamental differences.

1.3.1 Modeling, Discovery, and Monitoring of Resources

In all Grids, there is a need to describe the characteristics of the available resources using a general and extensible mechanism. Up to date information with sufficient level of detail is key to determine whether a given resource fulfills the requirements of a user. With a common information model in place, end user tools can focus on further information management issues such as to discover what resources are available and to monitor known resources. Discovery of available resources is normally performed by contacting an information index. Complete information about resources can either be aggregated into the index, or kept in a local information service on the resource itself. In the latter case, the index contains information about how to contact the local information service on the resource. In a typical resource discovery scenario, a client retrieves information about the set of currently available resources from an index, queries the most interesting resources for more detailed information, and selects which resource to use based on the retrieved information. A naive

way to perform monitoring is to periodically query the monitored resources for new information. Mechanisms to asynchronously notify interested parties with information updates are however preferred. Such notification mechanisms can reduce the number of messages significantly, allowing clients to retrieve updates as these are generated without resorting to extensive polling. While discovery includes the retrieval of both static and dynamic information, monitoring is exclusively used for dynamic resource information. A typical use case for resource monitoring is a client that supervises the progress of a task performed by a resource. The potentially very large number of users and resources in a Grid makes scalability and responsiveness important criteria for information management systems. For discovery, scalability is the main issue, as an index may have to serve a large number of clients. Responsiveness is more important for monitoring than for discovery, as up-to-date information is vital in some monitoring scenarios, e.g., fault detection.

Information management in Grids is complicated by the fact that information is more or less always old as the state of a Grid resource may change rapidly. The asynchronous communication mode of the general purpose Internet typically used to interconnect Grid resources makes it impossible to maintain an updated view of state in remote resources. Resource availability also varies, as resources at any time may join, or due to policy reasons or unforeseen events, disappear from, the Grid. To avoid stale index entries from no longer available resources, information indices should be self-cleaning via soft-state registrations. In addition to always being old, information is often incomplete, as resource owners are free to choose what resource information they publish for public access in a Grid. Resource information is hence not to be trusted blindly, as the presence of information about a resource in an index neither guarantees the availability of the resource, nor guarantees that a particular user is authorized to use it.

The existence of several VOs in a Grid results in a variable grouping of information with multiple, overlapping indices instead of one well-structured, hierarchical index structure containing all available information. Most indices contain general information about a subset of the resources in a VO, although specialized indices can be used, e.g., for keeping track of available storage locations and the amount of free space on each. Specialized indices can help to reduce the load on general purpose ones. It is however hard to anticipate every type of client information request, and it is not feasible to construct a large number of special-purpose indices. For performance reasons, users and other information consumers should instead be able to specify and limit the information they are interested in retrieving, e.g, through a query language such as XPath [45] or SQL [112].

1.3.2 Resource Allocation

In order to use a given Grid resource efficiently, mechanisms are needed to allocate the resource, to control and monitor the usage of it, and once done, to

release the allocated resource. As computational resources are the ones most commonly used in Grids, the rest of this section focuses on these, and the computational tasks (henceforth called *jobs*) that they execute. The described scenario focuses on the basic capability to execute a job on a predefined resource. Higher-level tasks, such as selection of which resource to run the job on, are discussed in Chapter 2.

There are many problems that need to be solved in order to provide high-quality job management mechanisms, both from a user and resource owner perspective. Users want a simple, secure, and efficient interface to initiate, monitor, and control jobs on a remote resource. Most existing Grid toolkits offer a set of basic functionalities that (partly) fulfills these requirements. A job description language allows users to express job configuration, e.g., the executable to run, job input and output files, as well as requirements on the resource executing the job, e.g., hardware architecture, amount of memory, and operating system. Job execution mechanisms typically provide an abstraction layer that hides the heterogeneity of the underlying execution platform. Batch system schedulers such as LoadLeveler [115], LSF [222], and PBS/Torque [110] are examples of such platforms. Other used execution backends include Condor [200], a loosely connected pool of machines available to Grid jobs when otherwise idle, and execution of the job on the frontend machine itself through the POSIX "Fork" command. Most job management mechanisms also define a state model for a computational job. Typical job states include "pending", "running", "finished", and "failed".

From a resource owner perspective, authentication and authorization mechanisms are needed to be able to control which users that may access the resources. Resource owners also want to control the environment a Grid job is executed in, e.g., via sandboxing techniques[1]. Other resource owner requirements include functionality for auditing and accounting, i.e., tracking who is using the resource, for what, at what time, and in what quantities.

More intricate aspects of resource (job) allocation such as interoperability of job submission systems and the associated standardization process, as well as negotiation of terms of use are discussed in Chapter 5 and Section 2.2.2, respectively.

1.3.3 Data Management

Many Grid applications require secure and efficient access to data that is stored distributed across a Grid. The GridFTP [61] protocol is the de-facto standard for transfer of data. GridFTP extends FTP, e.g., with authentication on both the data channel and the control channel, based either on the Grid Security Infrastructure [217] or Kerberos [152]. GridFTP contains several performance

[1]Sandboxing mechanisms provide a limited and secure execution environment for a not fully trusted application, in this case a Grid job. According to Thain et al. [200], sandboxing techniques must provide both the box, i.e., the protection mechanisms, and the sand, i.e., an execution environment as suitable as possible for the requirements of the application.

improvements over the original FTP protocol, including parallel transfers, i.e., the usage of multiple TCP data streams between two endpoints, and striped transfers, i.e., usage of data streams from different endpoints, both on the sender and receiver side. Other functionality improvements are restartable transfers and transfers of partial files. Third-party transfers (server-to-server transfers) allow a client to transfer files between two servers without acting as an intermediate proxy for the data channel. GridFTP also specifies both an option for automatic TCP buffer size negotiation and protocol messages for explicitly setting the buffer size.

In addition to the actual transfer of data, current Grid infrastructures typically contain higher-level data management capabilities. Reliable transfer of files is achieved by tracking progress of transfers and restarting these upon failure, as done e.g., by Globus RFT [8] and Stork [121]. The motive behind *data virtualization* is to decouple the file identity (the logical file name) from the location(s) of a file (the physical file name(s)). This is typically achieved by the usage of a catalogue, e.g., CASTOR [38] or Globus RLS [42], where mappings between logical and physical file names are stored. Virtualization of data enables replication [39, 64, 124], i.e., distribution of multiple physical copies of the same logical file across the Grid. Replication can improve both performance [120, 172] and fault tolerance [125, 130].

1.3.4 Security

As the resources used in a Grid typically are valuable and the data transferred between the resources may be confidential, security is an important aspect of Grid computing. New challenges arise in Grid security as the interactions between user tools and resources are more complex than the traditional client-server model. Grid security is further complicated by the fact that resources that belong to different administrative domains (trust domains) interact, each domain having different security policies and using different mechanisms to implement the respective policies.

Most Grid security scenarios are covered by three fundamental computer security concepts, often abbreviated AAA [212]: *Authentication* establishes the identity of other entity, in the Grid case, typically a user or a resource; *Authorization* concerns the privileges (access rights) to a particular entity (resource); and *Accounting* includes the control, monitoring, and metering (potentially including billing) of resource consumption.

From a user perspective, ease of use is key for a Grid security toolkit. A *single sign-on* mechanism allows users to authenticate themselves only once, instead of having to manually repeat the authentication procedure for each resource they interact with. The requirement for delegation of access rights arises from the often complex interaction pattern between users and resources. Through delegation, a user can grant a resource the permission to perform operations on behalf of the user. For resource owners it is vital that the Grid security mechanisms are easy to integrate with the local security infrastructure

used within the administrative domain. Resource owners also need mechanisms to track access to the resource, i.e., metering and accounting. Flexibility is the key issue for Grid application developers. A versatile API for authentication, delegation and similar tasks enables developers to cope with the complexity of the interactions between applications and Grid resources.

A commonly used security mechanism is the Grid Security Infrastructure (GSI) [81, 217]. GSI uses a public key infrastructure with X.509 certificates [105] and supports TLS (SSL) [57] for secure communication. Delegation and single sign-on are handled through proxy certificates [206], often simply referred to as proxies. A proxy certificate is valid for a short period of time only, typically a few hours.

1.3.5 A Basic Job Submission Scenario

A typical interaction between a user client and a set of Grid resources is illustrated in Figure 1. In this scenario, the user initially sends a query to the index to discover what resources are available. As described in Section 1.3.1, the configuration of the index determines whether the user retrieves all information available about the registered resources, or only references to other information sources that describe the resources in more detail. In the latter case, the user has to perform subsequent queries to find out more detailed information about the resources. After retrieving resource information, the user selects which resource to use and then submits the job to the selected resource by one of the job execution mechanisms described in Section 1.3.2. Finally, the user ensures that the job input files (including the executable) are transferred to the selected resource. This last task is either performed from the user client host via direct transfer, or, as illustrated in Figure 1, by the resource through third-party transfer. The presence of higher-level data management capabilities such as reliable file transfers and replication can improve both the performance and the fault tolerance of this part of the job submission process. Security mechanisms can be involved in all of the these tasks, including authentication of the user by the resource, authorization of the user's permission to execute the job, and delegation of the user's credentials to enable the resource to download job input files.

1.4 Outline

As described in Section 1.3.5 and illustrated in Figure 1, the basic Grid capabilities for management of information, jobs, data, and security, all provided by contemporary toolkits, can be used to perform most tasks in a basic job submission scenario. However, real-life Grid usage scenarios require capabilities beyond the fundamental ones described in Section 1.3. This thesis investigates topics that extend on the basic job submission scenario. The outline of the rest of this introduction is as follows. Chapter 2 describes architectural models

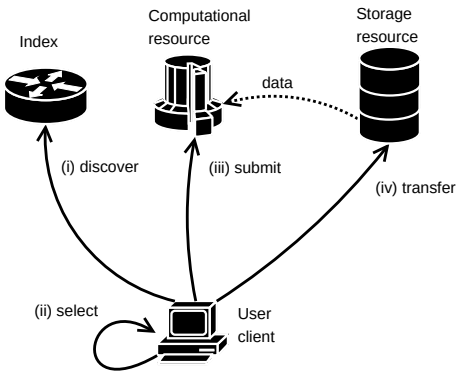


Figure 1: Component interactions in a basic job submission scenario.

and algorithmic considerations for resource selection, topics studied in papers I, II, and III. In Chapter 3, issues related to submission, monitoring, and fault handling of large numbers of independent jobs are studied. Frameworks for such job management capabilities is the topic of Paper IV. Chapter 4 describes workflows, i.e., sets of jobs with internal execution order dependencies. Chapter 4 also discusses issues related to composition of workflows, resource selection for jobs in a workflow, and workflow execution. Papers V, VI, and VII study a wide range of topics related to the composition, scheduling, and execution of Grid workflows. Chapter 5 describes interoperability problems in current Grid infrastructures and the related standardization efforts. This thesis addresses interoperability problems at several levels, e.g., at job submission level in papers I and II, job management level in paper IV, workflow execution level in paper V, and also from a more conceptual perspective for workflows in paper VII. Some design considerations for Grid software are given in Chapter 6, including discussions of service-oriented architectures, Web services, and design heuristics. A more in-depth analysis of these topics is found in Paper VIII. The final part of this introduction consists of a summary of the papers in the thesis (Chapter 7), an outline of potential future work (Chapter 8), and a bibliography, respectively.

Chapter 2

Grid Resource Brokering

The selection of which Grid resource(s) to use for a specific application is unfeasible even for the educated user due to the large number of resources, their fluctuating availability, differences in resource hardware, software, access policies, etc. A resource broker is a tool that automates and improves resource selection, or more commonly, the whole job submission process, for the user. A Grid resource broker is sometimes referred to as a meta-scheduler, as it selects which local scheduler (which local resource) to interact with.

2.1 Brokering Scenarios

There are two main Grid resource brokering scenarios. In the centralized scenario, all access to Grid resources is controlled by one broker. A centralized broker has good knowledge of, and control over, all submitted jobs and can, via load balancing techniques, produce good schedules. One obvious drawback of this type of broker is the potential performance and scalability bottleneck and that a centralized broker is a single point of failure. Another shortcoming of the centralized brokering approach is that it is hard to introduce dynamic policies, e.g., user-specified resource selection algorithms, in a centralized system. Furthermore, despite having complete knowledge about all the Grid jobs, a centralized broker cannot produce completely reliable schedules as the ultimate control over the resources remains in the hands of their respective owners.

The alternative approach is a decentralized (distributed) brokering architecture, where individual users have their own resource brokers. This type of broker typically manages only a fraction of the total number of jobs submitted to the Grid, and can hence not (alone) perform load balancing. Advantages of the distributed brokering approach include scalability and fault tolerance. A decentralized brokering architecture also enables customization, as each individual broker can be tailored to the specific requirements of an certain user or application. Centralized and distributed brokers, as well as hybrid brokering

approaches with a hierarchies of brokers, are further discussed in [122].

A centralized broker is typically used to schedule jobs to a specific set of resources, often belonging to the same VO. A decentralized broker on the other hand, is most often not tied to certain resources, but allows the user to specify for each job request which resources the broker should consider. There is however no fixed hierarchy of resources as some resources may be accessible to users from different VOs, possible even through different Grid middlewares. Such resources can be seen as belonging to multiple Grids.

The scheduling policy used by a resource broker can be either system-oriented or user-oriented [122]. The goal expressed in a system-oriented scheduling policy can be to maximize resource utilization, load balance, fairness, or a combination of these. System-oriented scheduling policies are commonly used by centralized brokers. A distributed broker most often strives to maximize a user-oriented scheduling policy, typically by improving the throughput or response time for jobs submitted by the individual user, regardless of the impact on the overall Grid performance and at the expense of competing users.

This thesis investigates the decentralized Grid brokering problem. The varying characteristics of different applications make resource selection a problem that must be solved on a per application basis. The problem is further complicated by heterogeneity in resource hardware, software and usage policies. Since a decentralized broker operates without global control, it must base all its decisions on information about, and negotiation with, the resources, and not on control over them. As discussed in Section 1.3.1, information gathered about the state of the available resources is often incomplete, as resources may limit the published information due to misconfiguration or policy reasons. Information is typically also outdated, as the current load of a resource may change at any time, making Grid resource brokering an online scheduling [166] problem. A decentralized broker may serve more than one user, but typically handles every user in isolation in such a scenario, in effect giving each user a personal broker. Users of a decentralized broker have to compete for resources with other users of the same broker, with users of other Grid brokers, and, as most resources are not dedicated to Grid use exclusively, with users accessing the resources through local, non-Grid interfaces.

All attempts to perform Grid resource brokering should consider the degree of transparency. Users want transparent access to the resources, but some issues, such as the resource selection criteria, typically requires some user involvement. A flexible mechanism that enables users to express their different resource requirements is of particular importance for a decentralized broker with a user-oriented scheduling policy, as user satisfaction is the main goal for this type of broker. A user's typical resource requirements for a computational job include hardware architecture and the number of CPUs of the resource, as well as the amount of main memory and secondary storage available to the job. Further resource requirements include operating system; installed software, including availability of licences for commercial software toolkits; capacity of the network connecting the resource to the Grid; and available QoS guarantees,

such as job start and/or completion time.

2.1.1 Approaches to Brokering

Figure 2 illustrates a decentralized brokering scenario with a set of computational Grid resources. Each resource uses a local batch system scheduler to plan and manage the execution of jobs submitted to that resource. The local schedulers control the backends that execute the jobs, see Section 1.3.2 for more details. Also shown in Figure 2 are two indices that store Grid resource information. The dotted arrows in the figure illustrate how resources register information about themselves in the indices. Resource information queries from brokers to the indices are shown as dashed arrows. For clarity, the two information indices aggregates all available resource information and no information queries are hence sent from the brokers to the resources. The two rectangularly shaped brokers in Figure 2 each serves a small set of users. The other type of broker, illustrated as Broker/Client in the right-hand side of the figure, is integrated into an application and is hence used by the user(s) of that application only. The solid arrows in Figure 2 illustrate job requests, either sent from the users to the brokers, or from a broker to a resource. Data transfers are omitted from the figure for clarity.

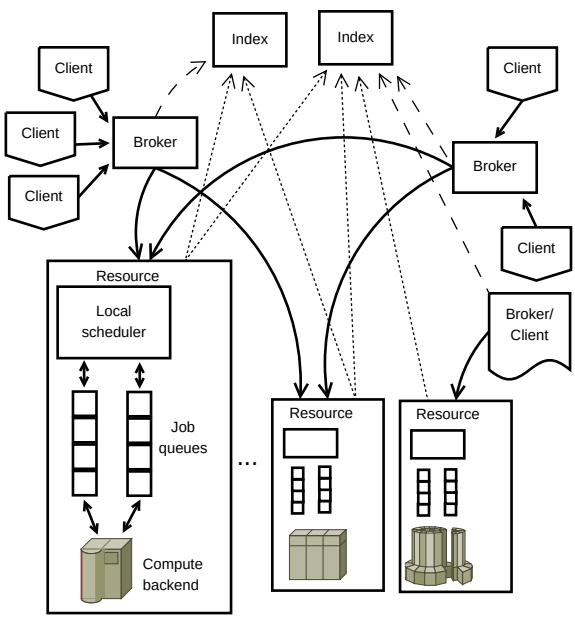


Figure 2: Overview of components and interactions in a decentralized brokering scenario.

General brokering discussions in the literature include "Ten actions when

Grid scheduling" [181], where Schopf discusses the eleven (!) steps in the job submission process. Pugliese et al. [168] analyze the requirements of Grid resource management and classify schedulers according to scheduling policy as well as the type and scope of resources they manage. Work by Vázquez-Poletti et al. [209] compares the centralized scheduling paradigm of EGEE WMS [63] with the decentralized one of GridWay [109].

Examples of resource broker software include EMPEROR [4], a meta-scheduler with a framework for implementing performance (prediction) based scheduling criteria. EMPEROR utilizes time series analysis to predict, e.g., job execution time and resource utilization. The composable ICENI [228] Grid scheduling architecture supports multiple scheduling algorithms, including random, simulated annealing, best of n random, and a game theoretic approach. The eNANOS Grid resource broker [176] supports submission and monitoring of Grid jobs and enables users to customize resource selection by weighting the importance of resource attributes such as CPU frequency and main memory size. Chapman et al. [40] present a Grid scheduling framework that uses prediction theory, in particular Kalman filters, to minimize response time for Grid jobs. A theoretical study of the Grid scheduling problem and a comparison to methods applied in multiprocessor systems is given by Schwiegelshohn et al. [182].

An alternative approach to resource selection is taken in market-based Grids, where participants trade resource shares in artificial markets. Claimed advantages of the marked-based approach are increased resource utilization and better load balancing, both a result of the supply and demand equilibrium that is predicted to occur in an economic system [221]. Users can furthermore assign priorities to their jobs by adjusting the budget allocated to each job. Examples of market-based mechanisms for Grid computing include preallocation of artificial Grid credits [87] and systems where resource consumption is based on real economic compensation [12, 107]. The pricing mechanism in a market-based Grid is often implemented either as a commodity market or as an auction. Additional models for market-based resource allocation exist and a taxonomy of these as well as a discussion of market-based systems can be found in the work by Yeo et al. [225].

Although the market-based approach may seem fundamentally different from resource selection algorithms used in other Grid systems, the task of a Grid resource broker remains much the same. The broker still has to identify, characterize, negotiate, select, etc. resources to solve the decentralized Grid brokering problem. The additional constraint in a market-based Grid is that the resource selection algorithm also has to consider price as a parameter. Instead of requesting the "best" resource, a user may prefer to use the best resource that fulfills certain budget constraints, i.e., resource selection becomes a tradeoff between cost and performance. This issue and other resource selection considerations are further discussed by Yahyapour [223].

2.2 Tasks of a Grid Resource Broker

As outlined in Section 1.3.5, the tasks performed by a Grid resource broker are discovery of available resources, selection of which resource to use, invocation of the job request operation of the chosen resource, and transfer of the job input files to the resource. Whereas most of these tasks are straight-forward, resource selection is a complicated process. In overview, resource selection consists of three phases. First, the list of discovered resources is filtered. In this step, resources are removed if they do not fulfill the user's requirements for hardware, software, etc, or if the user is not allowed to use them. After the initial filtering follows two dependent and often interleaved tasks, prediction of resource performance and negotiation of terms of use. Performance prediction includes estimating the characteristics of both the application and the resources considered. Resource heterogeneity results not only in performance differences between Grid resources, but also means that the relative performance characteristics may vary for different applications. This complicates predictions of job performance. Nevertheless, good performance estimates are of great value, e.g., during negotiation of terms of use. Such a negotiation of QoS terms result, when successful, in the creation of a Service Level Agreement (SLA) between the broker and the selected resource. Accurate performance predictions are also useful in cases when no performance guarantees are available. Various perspectives of performance prediction and SLA negotiation are discussed in the following sections.

Optional tasks for a broker include to monitor the job during execution and job clean up tasks. Jobs may also be migrated, e.g., for performance reasons, and resubmitted to alternative resources upon failure. Other commonly used features include management of sets of individual jobs and coordination of jobs with internal dependencies, the latter also known as workflows. Job management and workflows are further discussed in chapters 3 and 4, respectively.

2.2.1 Performance Predictions

An accurate estimate of job performance serves multiple purposes. During SLA negotiations, knowledge of application performance helps avoiding requesting more resource capacity than required, and can hence reduce response time, cost, and/or other scheduling criteria. Performance prediction techniques are also useful when no SLA negotiation occurs between a resource broker and Grid resources. In this case, job placement decisions are based exclusively on the (predicted) performance of jobs on the considered resources.

Research on performance prediction often focus on models for estimating various parts of the job lifecycle, including execution time and time spent waiting in a batch queue for access to the resource. This is a well studied problem for traditional batch system scheduling [187, 204]. In Grid environments, performance prediction is complicated by the existence of multiple heterogeneous machines, and the fact that the considered job need not have executed on all

of these before. One way of classifying performance prediction methods is to divide them into statistical methods that uses data gathered from previous executions [6, 116, 127, 187, 204] and heuristics-based methods that take into account job and resource characteristics [99, 128, 216].

Krauter et al. [122], divide performance prediction methods into predictive and non-predictive ones. The predictive methods, which include pricing models and machine learning, take historical use in consideration. Non-predictive methods include probability distributions that do not make use of historical data. In the taxonomy by Krauter et al., heuristical methods can be either predictive or non-predictive. Alternative classifications of performance prediction methods include work by Smith [185] that discusses statistical and analytical models. The statistical models are based on analysis of previously completed applications and equations are used to model execution time. The analytical ones are either derived by hand or by automatic code analysis or instrumentation [185]. Smith further describes two statistical run time prediction methods, investigates how to predict batch queue waiting time, and discusses scheduling methods that utilize run time predictions.

Ali et al. [7] analyze how to model execution time on heterogeneous computing systems and discuss the minimization of performance metrics such as job start time and job completion time. They present a model for expected execution time that takes into account heterogeneity of machines and tasks, as well as consistency, the latter defined as whether a given machine is faster than another one for all types of tasks. In [91], Goyeneche et al. evaluate the accuracy of current data mining and statistical methods for performance prediction based on application similarity classifications. Based on their findings, they propose a prediction mechanism with weighted templates that, after initialization, prediction, and incorporation of historical information, gives run time predictions with corresponding confidence intervals [91].

Smith [185] describes two techniques for batch queue wait time prediction. The first is based on run time predictions and simulates commonly used batch scheduling algorithms to transform the run time prediction problem into a queue time prediction one. The second method by Smith is history-based and uses application similarity categorization techniques to classify the scheduler and the application. Li et al. [127] apply local learning techniques [18] to queue time predictions. Evaluations using workflow traces from large clusters demonstrate that the local training techniques by Li et al. are more efficient than global and adaptive ones.

A prediction method that takes into account file transfer times, batch queue waiting time, and application execution time is suggested and evaluated by Smith [186]. The proposed method uses instance-based learning and is based on historical information.

2.2.2 Negotiation of QoS Terms

The term QoS refers to guarantees (beyond best effort) for the performance of a service. This covers virtually all aspects of service delivery (in this case job execution), including availability of the service, amount of resources allocated to the service request, compensation for the consumed resources, agreed levels of security, risk of failure, response time, throughput, etc. The essential mechanism to guarantee QoS terms is a priority mechanism that enables differentiation between consumers of a service. To support QoS in a general infrastructure, the following components are needed: a language to describe QoS terms, a protocol that consumers can use to negotiate a SLA with a provider, a mechanism to model the agreed upon SLA, and functionality to monitor the SLA for violations.

A language for describing QoS terms is by necessity domain dependent, e.g., bandwidth for networks, as defined by the Resource ReSerVation Protocol (RSVP) [31]. For computational jobs, typical QoS terms include amount of memory allocated to the job and number of CPUs, as well as job start and/or completion time. It is also useful to be able to express under what restrictions the terms apply, e.g., a resource will promise to complete a job before a certain deadline only if the user submits the job within a given time.

Various types of negotiation protocols have been studied within the area of agent-based computing. A framework for auction protocols is suggested by Chard and Bubendorfer [41]. Bai et al. [21] propose a three actor model with providers, brokers, and consumers. They evaluate an economic brokering algorithm with respect to resource utilization, consumer satisfaction and provider revenue. A game-theoretic approach is suggested by Khan et al. [207]. This work studies utilization, fairness, completion time, and request rejection rate for various cooperative and non-cooperative resource allocation methods. Venugapol et al. [210] propose a bilateral negotiation mechanism that, in addition to accept and reject messages, also allows the negotiating parties to express counter offers. Simulations confirm that this approach is beneficial for brokers that accept reservation start time delays. In context of the NextGRID project [46], Hasselmeyer et al. [101] discuss how to negotiate SLAs based on business-level objectives instead of details of the hardware used to deliver the service. For example, a user can select, e.g., from the "gold", "silver", and "bronze" service levels instead of specifying detailed XML descriptions of the requested resources.

A proposed standard that has received attention the last few years is WS-Agreement [14] that defines mechanisms to model and monitor agreements between a (Web service based) agreement provider and an agreement initiator. Seidel et al. [183] survey Grid resource management projects that utilizes WS-Agreement to model SLAs, typically to implement start time guarantees. A WS-Agreement-based negotiation protocol is proposed by Siddiqui et al. [184]. Here, resource allocation is modelled as a strip packing problem. The negotiation protocol is based on a three-layered approach, with allocators for single

requests, coallocators that coordinate requests from the same application, and coordinators that resolve conflicts between coallocators.

Advance reservations. Of particular interest for this thesis is response time related QoS aspects. In the batch system schedulers used in current Grid infrastructures, advance reservations is the only mechanism available to guarantee job start and/or completion times. An advance reservation is an assurance for the consumption of a certain amount of resources at a specified time in the future. However, such a guarantee may be cancelled due to resource failure or policy reasons, such as when a higher priority activity (typically another reservation) causes the reservation to be preempted. As advance reservations most often are given very high priority, the latter case is unlikely to occur. There are multiple uses for advance reservations, including time-critical tasks that must meet a deadline, which would be impossible without a start time guarantee. Further examples include reducing job start time uncertainty [141], demonstration purposes, debugging, and other interactive use, when access to the resource at a known time is critical.

Advance reservations also enable the job to be synchronized with other activities, which is essential for coallocation and workflows. The usage of advance reservations for computational resources however reduces resource utilization [138, 188, 189]. This reduction can be explained by increased fragmentation of batch queues that in turn reduces the efficiency of the backfilling scheduling algorithms used in current batch systems. Castillo et al. [37] use methods from computational geometry to tackle fragmentation. Another possibility to achieve good performance despite the use of reservations is to allow a certain degree of laxity in reservation start times [71]. However, current batch system schedulers lack the mechanisms required to implement such reservation rearrangements [134]. Qu describes an architecture [169] where advance reservations can be offered to Grid jobs regardless of whether they are supported at the local batch system level. This is achieved by adding a reservation management layer in the Grid scheduler. However, this solution assumes that no jobs are submitted to resources through non-Grid interfaces or by other resource brokers.

Coallocation. Another QoS related topic studied in this thesis is coallocation, i.e., the coordinated allocation of a set of resources to be used together for solving a problem. A typical use case for coallocation is when multiple parallel computers are used to execute a job that communicates not only between the CPUs in one cluster, but also across different machines, using e.g., the Message Passing Interface (MPI) [145]. A more complex scenario is the concurrent usage of instruments and computers to in real-time gather, analyze, and possibly visualize, scientific data from an experiment.

In order to provide a coallocation mechanism, a resource broker must solve two problems. The first problem is the selection of which set of resources to use. This is a non-trivial selection procedure as the set of resources suitable

for the subjobs forming the coallocated job may overlap, as happens, e.g., in the MPI-scenario where multiple instances of an application must execute on a set of (binary compatible) machines. The second problem is to ensure that the selected resources have a coordinated (most often common) start time. This is typically guaranteed through the use of advance reservations. In addition to the two problems related to the selection and allocation of the resources, a coallocation scenario also requires initialization and synchronization of the coallocated subjobs. For many applications, e.g., jobs that use cross-cluster MPI, each subjob must be aware of, and able to communicate with, the other subjobs. Preferably, this functionality should be as transparent as possible to the application. For the MPI scenario, Coti et al. [48] suggest a solution that uses connectivity services to hide the complexity of communication.

The work by Czajkowski et al. [50] addresses the application initialization problem by defining a library for initiating and controlling coallocation requests and another library for application synchronization. By compiling an application that requires coallocation with the application library, the subjob instances can be instructed to wait for eachother at a barrier prior to commencing execution. A similar project is the Globus Architecture for Reservation and Allocation (GARA) [80] that provides a programming interface to simplify the construction of application-level coallocators. GARA can perform both immediate reservations (allocations) and advance reservations of networks, computers, and storage.

Attempts to solve the second coallocation problem, start time coordination, include the KOALA system [147] that uses a mechanism for implementing coallocation without using advance reservations. This is achieved by requesting longer execution times than required by the jobs, and delaying the start of each job until all allocated jobs are ready to start executing. Another contribution where coallocation is achieved without the use of advance reservations is [19], where Azzedin et al. propose an algorithm based on synchronous queueing. MacLaren [135] treats coallocation as a transaction problem and uses the PAXOS commit protocol to ensure consistency. This protocol is based on messages that create, modify, and cancel reservations. Mateescu [139] defines an architecture for coallocation based on Globus Toolkit 2. The suggested coallocation algorithm uses a window of acceptable job start times and tries to reserve all required resources at predefined positions in this start time window. Decker et al. [51] describe another window-based coallocation algorithm that by considering the execution and communication times of a set of dependent tasks tries to minimize the overall completion time. The coallocation algorithm suggested Wäldrich et al. [215] models reservations using the WS-Agreement framework and uses a concept of coallocation iterations. In each iteration of the algorithm, a list of free time slots is requested from each local scheduler. Next, an off-line matching of the time slots with the resource requests is performed. If the complete coallocation request can be mapped onto some set of resources, reservations are requested for the selected slots. Röblitz et al. [175] describe a coreservation architecture where resources are matched to coreserva-

tion requests in a off-line style similar to the algorithm by Wäldrich et al. [215]. The coreservation algorithm by Röblitz et al. can handle both temporal and spatial dependencies between requested resources. Netto et al. [151] discuss a coallocation algorithm based on malleable (flexible) reservations and processor remapping. In this work, coallocation requests can be moldable, i.e., the number of requested resources can change. Individual resource requests can also change in terms of request size and/or start time, all in order to achieve the earliest possible start time. The actual assignment of resources to requests is performed with an off-line matching algorithm. A coscheduling (coallocation) mechanism inspired by the coordinating tasks performed by a travel agent to guarantee the availability of a multi-resource itinenary is suggested by Yoshimoto et al. [227]. In ASKALON [184], coallocation is modelled as a constraint-satisfaction-problem. An on-line coallocation algorithm described by Castillo in her PhD thesis [36] makes use of tree structures to handle free time slots at the resources and decides which resources to reserve by traversing the trees.

2.3 Non-goals of a Decentralized Broker

In the discussion of the tasks of a decentralized broker, it is useful to explicitly state some related tasks not performed by this type of broker, and discuss why these tasks are best handled by other components.

A good distribution of the load over the resources in a Grid is important for both achieving good performance and utilizing the resources efficiently. A decentralized resource broker that only handles a small fraction of the total number of submitted jobs cannot alone achieve good load balance. However, as decentralized brokers typically seek to minimize job response time, each broker contributes to the overall load balance of the Grid by avoiding the most heavily loaded resources.

One capacity allocation mechanism commonly used in scientific collaborative Grids is policy-driven preallocation of resource shares based on scientific impact. Fairness in this type of Grid can for a given user be described as the difference between the user's historical, and possible also current, resource usage and the preallocated amount of resources the user is entitled to [66]. A decentralized resource broker does not necessarily try to enforce fairness. On the contrary, the broker could be considered successful if it manages to allocate more resources than the user is entitled to. Fairness should hence be enforced on the resource side [66], not by throttling mechanisms in the broker. A topic related to fairness is accounting [87], i.e., book keeping the amount of resources consumed by a user. As fairness, possible payment, etc. typically are determined from accounting information, accounting should be the responsibility of the resource provider, not the consumer (i.e., the broker). A user trying to circumvent accounting (and hence also payment and fairness) could otherwise use an alternative broker that does not report resource consumption truthfully.

2.4 Contributions to Grid Resource Brokering

The results related to resource brokering in this thesis is the definition of the total time to delivery for a Grid job and an associated unified model for resource selection that can utilize, but do not depend on, a wide range of prediction techniques as well as SLA negotiation mechanisms. This work also demonstrates how advance reservations can be used both to meet hard deadline constraints and reduce uncertainty in resource selection. Algorithmical contributions in the thesis include advances to the state-of-the-art in coallocation.

Chapter 3

Grid Job Management

Although a Grid resource broker handles the perhaps most cumbersome operation - resource selection, it is not a complete turnkey solution for Grid enabling an application. Other capabilities typically required by Grid application developers include monitoring of submitted jobs, resubmission of failed jobs, and potentially also migration of jobs from slowly or non-responding resources. Complex applications such as computational steering and interactive visualization need, in addition to job monitoring, also the ability to interactively steer the job during its execution.

One common problem type that is particular suitable to Grid environments is the parameter sweep study, where large numbers of jobs without internal dependencies are executed, typically to study a problem for different input parameters. Parameter sweep applications require tools that simplify the management, and execution (as well as reexecution) of large groups of jobs. Performance critical applications may need performance-aware job monitoring, and also a framework that automatically detects performance anomalies and takes appropriate actions, e.g., submits the job to an alternative resource, or, if possible, migrates the running job. Performance aware job migration requires both a checkpointing mechanism and a method to communicate application-specific performance metrics from the application to the Grid middleware. As both checkpointing and performance metric reporting necessitate application modification, some users are reluctant to use these features.

Some resource-demanding users use the Grid as their daily production environment for running very large numbers of computationally intensive applications. These users would benefit from having their own personal job queues, where jobs can be added for later submission to the Grid. To avoid overloading the Grid resources, user job queues typically include some sort of backoff functionality. Ideally, such functionality should not be required. However, the job submission pattern of some users shows a temporal and spatial burstiness [129], and jobs failure due to resource overload is hence not uncommon [129]. Personal job queues also enable inter-job priorities for jobs that belong to the same

user. This allows users, to some extent, to control the order in which their jobs are executed. The exact job execution order is however determined by batch system scheduling algorithms and the user can hence not control the internal execution order of jobs after they are submitted to batch queues.

3.1 The Role of Job Management Frameworks

Many Grid application developers implement their own set of tools for the above described functionalities. This approach may at a glance seem attractive as it potentially can save time if only trivial job management functionality is needed, but typically results in non-reusable software due to too strong coupling to the application. Furthermore, the built-in heterogeneity of Grid infrastructures and their error-prone nature necessitate middleware-independent job management tools and robust fault tolerance mechanisms that are non-trivial to design and implement. Preferably, resource brokering and job management capabilities as those described here should be gathered in well defined APIs, or as argued in Section 6.1, exposed as services.

3.2 Contemporary Job Management Tools

The construction of general toolkits for Grid job management is a large area of active research. Casanova et al. [35] describe a user-level middleware with an integrated scheduler (resource broker) that simplifies the execution of parameter sweeps on a Grid. The scheduling algorithm used combines Gant charts with various heuristics for estimating and minimizing completion times of jobs as well as of file transfers. The Nimrod-G [2] resource broker consists of a task farming engine, a scheduler (for resource discovery, trading, and scheduling), dispatchers and actuators (for interfacing different Grid middlewares), and agents for managing job execution, e.g., setting up the job environment on resources. Nimrod-G supports parameter sweeps and master-worker style applications. The goal of the GridWay [109] framework is simplified and more efficient job execution in a "submit and forget" fashion. The GridWay resource selection algorithms strive to minimize the job completion time. In GridWay, a Grid-aware application contains both a performance profile and a restart file, the latter used for user-level checkpointing. GridWay supports job resubmission, which can be initiated by failures, user requests, or performance declines. The last two scenarios are enabled by the checkpointing mechanism. Users of GridWay have their own personal submission agents (job queues).

The GridLab Grid Application Toolkit (GAT) [9] aims to provide a simple and robust environment for developing applications that exploit the Grid. GAT provides a layered view of the available functionality, ranging (bottom up) from a core layer, a service layer, a GAT API layer, and finally an application layer. The GridLab Resource Management System (GRMS) resides at the service layer. It provides services for job management, and management of

infrastructure elements such as information, data, networks, and accounting. Job management capabilities include a Job receiver (job queue) and services for resource discovery, job requirement prediction, resource performance estimation, and brokering, as well as mechanisms for negotiating QoS terms and performing advance reservations. The GridSAM [144] job submission pipeline is based on the principle of a staged event-driven architecture [218]. Instead of treating each job submission in isolation, the event-driven job submission pipeline is divided into stages, allowing the processing load of each stage to be controlled by adjusting the size of a thread pool serving it, thus increasing fairness under high load. Steps in the GridSAM pipeline include file stage-in, job description generation, job submission, job monitoring, file stage-out, and cleanup.

One main focus of the P-GRADE project [114] is to run parallel applications, i.e., jobs using MPI or PVM, on the Grid. P-Grade provides a graphical environment to design, execute, and monitor applications. The design of message passing applications is facilitated through a graphical language that enables also non-expert users to define the communication pattern between the processes. In P-Grade, checkpointing (and hence also migration) is supported for PVM applications. The focus of the Falkon system [170] is to improve performance for large numbers of submitted jobs. In Falkon, a provisioner creates and destroys executors. These executors are placed on available Grid resources and can accept jobs submitted by the user. This model provides for good performance and is particularly well suited for fine-grained jobs as the non-neglectable overhead for submitting a task to a Grid resource is done only once for each executor, instead of once for each job. Moltó et al. [148] discuss how to model and implement a resource brokering architecture using the Web Services Resource Framework (further discussed in Section 6.1). Their architecture is divided into three layers, information management, job submission, and metascheduling of multiple tasks.

Closely related to Grid job management is the process of enabling a certain application to execute on the Grid. By wrapping legacy code applications as Grid services, GEMLCA [56] provides submission, monitoring and result retrieval of jobs that execute legacy codes. The GEMLCA approach to wrapping the applications is non-invasive, as it does not require access to, or modification of, the application source code. Users can deploy new applications into the GEMLCA hosting environment through a Web portal and also manage jobs executing these applications. Mateos et al. [140] discuss various approaches to gridify, i.e., Grid-enable, applications. In their work, a set of existing frameworks are compared with respect to job granularity and ease of gridification, the latter including aspects such as whether the application source code needs to be recompiled.

3.3 Contributions to Grid Job Management

The main contribution to Grid job management in this thesis is the investigation of suitable architectures for job management tools. Our study proposes a multi-layered architecture in order to improve abstraction and flexibility. A proof-of-concept implementation demonstrates the feasibility of this model from a performance point of view.

Chapter 4

Grid Workflows

A *workflow* is a set of coordinated activities that combined solve a complex problem. The tasks that constitute a workflow are typically arranged as nodes in a directed graph where the links represent dependencies between tasks. These graphs are typically acyclical (so-called DAGs), although some workflow systems use graphs with cycles in order to model iterations.

The edges in the workflow graph can denote a *control flow* that explicitly dictates the execution order of the workflow tasks. This model of computation shares many characteristics with sequential programming languages and can hence easily be understood by programmers. Alternatively, the workflow graph edges can specify a *data flow* that implicitly defines a partial execution order of the workflow tasks. In the data flow model, a workflow can be thought of as a flow of data, where the nodes (tasks) merely are functions that transform the data as it flows through the graph. Although perhaps less straight-forward to most programmers, the data flow model has proven useful for non-programmers and application scientists [162]. Other Grid workflow approaches [104] use *Petri nets* as model of computation. Petri nets are a Turing complete formalism commonly used for modeling and analyzing systems that are concurrent, asynchronous, and non-deterministic.

Workflows are not exclusive to Grid computing, research on these started long before Grids existed. The emergence of Grid computing enables computational workflows to leverage powerful Grid resources for computationally intensive tasks. Such Grid workflows typically consist of a combination of computational jobs and file transfers between the machines that execute the jobs. A broad introduction to Grid workflows as well as a in-depth description of contemporary workflow systems is found in the book by Taylor et al. [196]. Van der Aalst et al. [208] discuss commonly used patterns in workflows. Taxonomies of workflow capabilities include [53, 230]. For a wide overview of current research in scientific (Grid) workflows, see e.g., [25, 55, 85, 133, 196].

4.1 Workflow Management Capabilities

The typical tasks in the scientific workflow process include composition of a new workflow, scheduling of the workflow, workflow enactment (execution), and analysis of the results, the latter including recording of enactment metadata for reproducibility purposes.

4.1.1 Workflow Composition

Important aspects when composing a new workflow include the language used to encode the workflow, the tools available to design a particular workflow, and to what extend reuse of existing workflows are possible.

The composition of a scientific workflow is simplified if the workflow language has expressive power equivalent to that of a (Turing complete) programming language. However, there is disagreement within the workflow community about whether workflow languages should be simpler and more restricted than programming languages [52], or whether constructs such as iterations and conditions are essential to encode complex scientific problems [20].

As most workflow systems use a textual, often XML-based, representation of the workflow tasks and their dependencies, design of a new workflow can be performed by encoding the workflow description manually. This can however be error-prone even for expert users, and the usage of a drag-and-drop style GUI client for defining the workflow graph is common.

Workflow composition is greatly simplified if existing workflows can be reused in parts or in full. This is exemplified by the Taverna [158] tool, where more than 3000 bioinformatics services can be utilized in new workflows. The myExperiment project [179] allows scientists to find, share, modify, and reuse workflows, as well as build communities and collaborate with colleagues in a manner inspired by social networking Web sites.

4.1.2 Workflow Scheduling

Workflow scheduling is the transformation of an *abstract workflow* with only tasks specified, to a *concrete workflow*, where resources have been assigned to each task. As a Grid workflow consists of a combination of computational tasks and file transfers, all considerations that apply for the brokering and management of individual jobs (as discussed in chapters 2 and 3) are also valid for workflow scheduling. The main difference between workflow scheduling and management of independent jobs, is to what extent workflow scheduling system should take dependency constraints among the tasks in the workflow into account. One method is full-ahead planning, where resources are assigned to each task at the start of the workflow execution. The completely opposite approach is just-in-time workflow scheduling, where each task is scheduled as its preceding task(s) complete. Hybrid solutions also exist, where the workflow is divided into subgraphs that are planned as execution of previous subgraphs complete.

Full-ahead planning techniques typically involves determining the *critical path* of the workflow, that is, the path of subsequent tasks in the workflow graph with the longest estimated total execution time. By giving higher priority to the tasks along this path, critical path workflow scheduling algorithms aims to minimize the workflow completion time, the so called *makespan*. However, just-in-time scheduling may also reduce the makespan, as this technique can leverage the dynamic nature of the Grid, and benefit from appearance of new resources and/or changed load on known ones. To minimize execution time, the workflow scheduling system can also rewrite the workflow graph, e.g., to group tasks for increased data reuse.

The Pegasus workflow management system [54], is an example of a workflow scheduler that reorganizes the DAG during the mapping process. In Pegasus, the final concrete DAG is produced in a format interpretable by an enactment engine such as Condor DAGMan [200]. Pegasus supports the use of placeholder jobs that in advance are submitted to Grid resources for execution. Once running, a placeholder job can execute one or more tasks in the workflow. When combined with task clustering, placeholder jobs can improve the workflow performance considerably, especially for fine-grained tasks [54]. Huang et al. [108] propose a simplified workflow scheduling algorithm that takes into account neither resource information nor task information beyond intertask dependencies. In this work, resources are grouped by heterogeneity and connectivity. A performance evaluation confirms that their proposed method gives shorter makespans than critical path algorithms.

Before the emergence of Grid workflows, the problem of scheduling of a set of dependent tasks, typically expressed as a DAG, on a set of heterogeneous processors, has been studied extensively. Examples of algorithms include Heterogeneous Earlist-Finish-Time (HEFT) [203], that also has been evaluated in the context of Grid workflow scheduling [219]. Rotithor's taxonomy [178] of dynamic task scheduling schemes is based on a separation of state estimation (i.e., resource load and/or performance prediction) and decision making. Alhusaini et al. [5] describe a DAG scheduling algorithm to minimize the makespan that operates on one layer of the DAG at the time. In this work, various heuristics for scheduling of tasks from a given layer are evaluated, including min-finish-time and max-finish-time. In the work by Canon et al. [33], 20 DAG scheduling heuristics are evaluated, focusing on makespan and robustness of the heuristics, the latter considering to what extent they are affected by unpredictable run-time changes.

4.1.3 Workflow Enactment

Grid workflow enactment can be performed using different levels of abstractions. For job-based workflows, some systems interact directly with batch system schedulers, whereas others use a Grid middleware, or even a toolkit that abstracts over multiple middlewares, to manage the workflow tasks. Alternatively, if the workflow is composed of a set of orchestrated services, workflow

enactment is the process of invoking the services in the specified order. As failures are frequent in Grid environments, fault tolerance is an important aspect of workflow enactment. This can be achieved on a per-task basis, e.g., by re-executing a failed task on an alternative resource or restarting an interrupted file transfer with a replica of the transferred file. These task-level fault recovery mechanisms are in the literature described as implicit fault management [104]. Workflow-level fault tolerance is another possibility, where an alternative path of enactment in the workflow graph is taken upon failure. However, the latter approach requires that the workflow language supports conditions and/or exceptions. A comparison of fault tolerance techniques in contemporary workflow systems is provided by Plankensteiner et al. [167]. As only concrete workflows can be enacted, there is a strong coupling between tools for workflow scheduling and workflow enactment, especially if workflow scheduling is performed in a just-in-time fashion (as opposed to full-ahead planning).

One commonly used workflow engine (workflow enactment system) is the control flow based Condor directed acyclic graph manager (DAGMan) [200]. In DAGMan, a workflow graph consists of computational tasks (Condors jobs) along with pre and post scripts, that can be used e.g., for data transfers. DAGMan is used as enactment engine, e.g., by Pegasus [54] and P-GRADE [89, 114]. The goal of the MOTEUR workflow engine [88] is to combine ease of use through simplified iterations (parameter sweeps) with flexible data composition mechanisms and efficient parallel execution. The data composition strategies in MOTEUR enables users, through the Scufl language of Taverna, to express nested dot and cross products over data sets.

4.1.4 Workflow Reproducibility Concerns

Reproducibility is key in the scientific process. In order to verify and repeat experiments, scientific workflow systems need to store information about input data sets, used Grid resources, versions of softwares and libraries, etc. This *provenance* metadata enables, when properly recorded, the workflow execution to be reproduced. Ideally, the workflow provenance information should not only capture the computational steps of the Grid workflow, but rather be a complete description of the greater, scientific process of generating new knowledge. Simmhan et al. [226] discuss various provenance issues as well as survey provenance systems and workflow management tools that support provenance.

4.2 Contemporary Grid Workflow Systems

Well-known scientific workflow systems include the ASKALON Grid application development and computing environment [69, 70]. The ASKALON architecture includes a Scheduler for placement decisions, a Resource Manager that performs resource discovery and related tasks, and an Enactment Engine for reliable and fault tolerant execution of tasks. The UML-based AGWL lan-

guage used to define workflows supports both control and data flow, as well as conditions and iterations. In ASKALON, placement decisions are based on a combination of advance reservations and performance predictions. The ASKALON scheduling process is divided into refinement (workflow rewriting), mapping (resource to task matching), and, upon failures or similar events, rescheduling. The used scheduling algorithms include HEFT.

Taverna [157] is an extensive workflow environment for the life science community. Workflows in Taverna are expressed in the Scufl language that supports both data flow and control flow. Taverna users benefit from being able to reuse a large set of predefined workflow services. Notably, as users only select among services hosted at predefined locations, no workflow scheduling is performed in Taverna.

The Kepler [132] workflow management system defines a data flow model where *actors* modify the data. Data transformation actors can fine-tune the data streams using, e.g. XSLT [58] or XQueury [29]. Kepler supports a wide range of control and data flow enactment styles, including synchronous communication, publish-subscribe notifications, and continuous time feedback loops. In Kepler, parameter sweeps are implemented using higher-order functions.

Triana [44, 197] is fundamentally a data flow system, but supports control flow through trigger tokens. Triana uses a component-based execution model, similar to that of the Common Component Architecture (CCA) [73]. Components can either be job-based or service-based. The job-based workflows execute on the Grid through the GridLab Grid Application Toolkit (GAT) [10], with bindings to the Globus GRAM, GridFTP, etc. Such workflows can be scheduled using the GRMS [123] described in Chapter 3. Triana supports service-based workflows by orchestrating services using the GAP toolkit [198]. There are no iterations or condition constructs in the Triana language, these are instead embedded in custom components. Triana decouples the GUI for workflow design from the workflow enactment system.

The P-GRADE [114, 89] Web portal allows users to define and manage workflows. Similar to Triana, the P-Grade portal supports (through DAGMan [200]) enactment of workflows composed of tasks, or of services, the latter through MOTEUR [88]. Nodes in a P-GRADE workflow graph can either be jobs or GEMLCA legacy code services. P-GRADE allows the use of resources from multiple Grids, and uses MyProxy [23] to enable the use of multiple certificates for the same user. Parameter sweeps are supported through parameter spaces that are mapped into data segments and further on to tasks (DAGMan jobs or MOTEUR service invocations) in the workflow manager. This mapping can either be dot or cross product.

Karajan [213], the workflow component of the Globus Java CoG kit, is based on earlier work on GridAnt, a Grid extension to the ant build system [17]. The Karajan system supports job execution and file transfers, as well as monitoring and checkpointing of workflows. In Karajan, a declarative control flow language is used, with constructs for expressing sequential and parallel tasks as well as conditions and iterations, where the latter can be either sequential or parallel.

Construction of large workflows is simplified by the support for user-defined procedures that enable custom extensions to the Karajan language.

The Imperial College e-Science Networked Environment (ICENI) [141, 142] workflow pipeline consists of the three phases: specification, realization, and execution. ICENI provides a pluggable scheduling framework that easily can incorporate new algorithms. Rapid completion of time critical tasks is ensured either by advance reservations or by placeholder jobs, the latter a mechanism also used by Pegasus [54] and Falkon [170].

4.3 Contributions to Grid Workflows

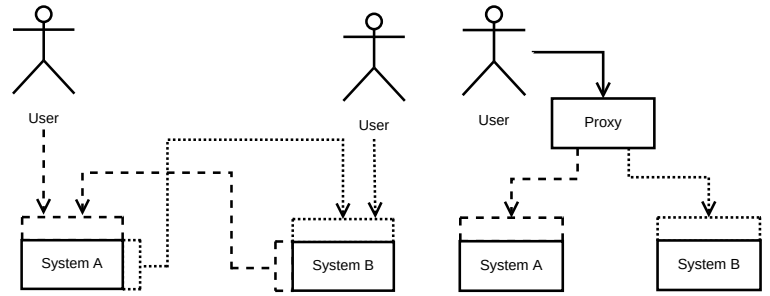
This thesis includes studies of suitable abstractions, techniques, and tools for integrating local and Grid resources in scientific workflows. One result is the investigation of suitable programming models for parameter sweeps and tools that enable the binding of workflow task parameters to be delayed until enactment time. We also survey workflow interoperability, albeit with a different approach than the standards-adoption one taken for resources brokering and job management. Our interoperability study has a theoretical perspective with particular focus on differences between Grid workflows and locally executing ones. This study builds on research within the areas of theory of computation, compiler optimization, and visual programming languages.

Chapter 5

Grid Interoperability and Standardization

There is today a somewhat paradoxical situation where Grids, partly being developed to increase interoperability between different computing platforms, themselves to high extent have interoperability problems. Although the reasons are obvious, expected, and almost impossible to circumvent (as the task of defining appropriate standards, models, and best practices must be preceded by basic research and real-world experiments), it makes development of portable Grid applications hard. The current situation with isolated islands of Grid infrastructures unable to interoperate with eachother is partly due to the many Grid projects that focus on infrastructures tailored for specific application areas despite only slightly different use cases. These projects have resulted in the creation of a number of Grid middlewares with similar functionality but without the ability to interoperate. The problem of achieving interoperability is however not only a technical one, but also a policy (political) issue. Lack of coordination among Grid projects and the related research communities have resulted in fragmented infrastructures without a clear structure. There is furthermore large overlaps in existing virtual organizations. A user can belong to multiple VOs, and be identified by different certificates in these VOs.

Although interoperability intuitively can be understood from a high-level perspective, the concept has no single agreed upon definition. IEEE defines interoperability as "The ability of two or more systems or components to exchange information and to use the information that has been exchanged" [111]. This definition captures the intuitive understanding that the interoperable systems must be able to communicate in a meaningful way. Another definition, by ISO, describes interoperability as "The capability to communicate, execute programs, or transfer data among various functional units in a manner that requires the user to have little or no knowledge of the unique characteristics of those units" [72]. One important aspect of this definition is that the use



(a) Gateway approach to interoperability. (b) Proxy approach to interoperability.

Figure 3: Different approaches to interoperability through user transparency.

(i.e., communication, execution, data transfer etc.) of two interoperable systems should be transparent to the client, be it a human operator or another system. For this second definition, there are (at least) two possible approaches to achieve the envisioned transparency. In the gateway approach, illustrated in Figure 3(a), one or both systems are modified to communicate with the other. The end user invokes the original system through its native interface, and requests are transparently forwarded to the second system. This transparency increases usability, but can on the other hand be seen as intrusive, as the gateway approach necessitates modification of both systems. The second option, shown in Figure 3(b), is to add an additional layer (a proxy) on top of the two systems that hides the differences in operation. With this approach, the end user interacts with the proxy layer, instead of with the individual systems. The benefits of this is that no modifications are required to the existing systems, which both simplifies implementation and reduces intrusiveness. However, the wrapping layer may appear different to the user than the native system interfaces, and can hence be more difficult to use.

In achieving interoperability for Grid middleware, the proxy approach is more feasible as it can be implemented on the client side. The gateway approach necessitates modification to software running on the resource side. This is impractical, and potentially also problematic from a policy point of view, as resource owners not necessarily agree that their machines need modifications and/or reconfigurations.

5.1 A Layered View on Interoperability

One well-proven method to achieve interoperability is to agree on standardized protocols, interfaces, and data formats. Standardization of lower-level protocols and data formats enables the definition of other ones with a higher level of abstraction. This can be exemplified by the successful Internet network stack,

where higher-level protocols such as HTTP and FTP are constructed on top of lower-level ones such as Ethernet and IP. A data format example of this principle is SOAP that leverages XML and XML schema. This idea of hierarchical reuse can be observed also in Grid computing. Within the data management area, the GridFTP protocol, further discussed in Section 1.3.3, greatly simplifies the construction of higher-level data services such as Stork [121] and RFT [136].

Unlike the GridFTP case, where a well-proven standard (the FTP protocol) existed, early Grid job management functionality was developed as a distributed extension to non-standardized batch system interfaces. These batch system interfaces were much later standardized through OGF DRMAA [171]. One commonly used early job management protocol is the Globus GRAM [76], which later was redesigned with Web service tools and renamed WS-GRAM. Although adopted by many projects as foundation for higher-level job management components, the GRAM interface never became a standard, and alternative implementations of the offered functionality exists, as illustrated by the NorduGrid/ARC middleware [65] that performs job management through GridFTP. The OGF has recently promoted two specifications that are important for standardization of job management functionality, JSDL [16], a language to describe job requests, and an interface to the job submission system, OGSA-BES [79]. Although rather new, both of these have gained widespread support [143].

In both the data management and job management cases, one can observe that standardization only occurs up to the point when there no longer exists a broad agreement on the required functionality and/or semantics. For data management, agreement exist on the transport level (GridFTP), but consensus has yet to be established, e.g., for replication and fault-tolerant transfers. Participants within the job management community agree on the language (JSDL) and the basic submission mechanism (OGSA-BES), but still struggle to define higher level services such as negotiation and resource selection. Some conceptual ideas are specified in the OGSA Resource Selection Services RSS [83], where the Candidate Set Generator (CSG) identifies resources where a job can execute, whereas the higher level Execution Planning Service (EPS) decides where a job should execute. The complete interfaces of these services, as well as their exact semantics remain to be defined.

The interoperability problems within contemporary Grid environments are the motivation behind substantial efforts from both research and infrastructure projects. These efforts can be divided into two classes, short-term solutions to provide interoperation between existing infrastructures and more sustainable approaches based on adoption of standard data formats and interfaces. The focus of the Grid Interoperability Now (GIN) project (later renamed Grid Interoperation Now) [96] is to provide increased interoperation among existing Grid infrastructures. Examples of software tools that interoperate with multiple middlewares include GridWay [109], which interfaces resources running one of GT2, GT4, and LCG. Venugopal et al. [211] share their experiences

in designing a broker that interoperates with various batch systems as well as multiple Grid middlewares. In an approach suggested by Kertész et al. [118], interoperability is achieved on a resource broker level instead of a Grid middleware level. Rather than interfacing resources directly, their software interacts with the respective resource brokers of the used middlewares. Kertész et al. also define a language for communicating broker capabilities [119]. A similar approach is suggested by Rodero et al. [177] that argue in favor of a meta-broker based on their experiences from the HPCEuropa project. The proposed meta-broker would have unified mechanisms for all tasks in the job management process and interact with Grid-specific brokers. Rodero et al. also survey to what extent existing and proposed standards fulfill the requirements of the meta-broker. Bobroff et al. [30] discuss different architectures for interoperable metascheduling and present a hybrid approach that combines hierarchical and peer-to-peer architectures. The contribution by Peirantoni et al. [165] utilizes WSRF-based Metagrid services as a bridge between users and different Grids. Set theory is used to formally describe the Metagrid services and their interactions.

Notable examples of projects that focus on standardization include the Genesis II [150] Grid system. Genesis II supports a wide range or OGSA standards, including JSDL, OGSA-BES, OGSA Resource Namespace Service (RNS) [164], and OGSA-ByteIO [149]. A strong focus on usability allows users to interact with Genesis II resources through a set of common shell commands such as, e.g., cat, cp, and ls. An OGF report [143] shares implementation and interoperability experiences of JSDL and surveys projects that adopt this standard. An OGF report [153] by Newhouse et al. discuss five scenarios related to the execution of applications on a cluster from an interoperability perspective. This report further illustrates how these scenarios can be realized using OGSA standards such as JSDL (with proposed extensions), OGSA-BES, the GLUE 2 information model [13], RNS, and DRMAA.

The purpose of an ongoing collaboration between the European Telecommunications Standards Institute (ETSI) [68] and the OGF is to ensure that the standards for communication within the Grid are based on requirements from both industry and research. The Open Middleware Infrastructure Institute (OMII) Europe [159] targets to make components for job management (OGSA-BES), data integration, and accounting available for multiple platforms, e.g., gLite [62], Globus [75], and UNICORE [195]. The UniGrids project [174] focuses on developing an OGSA-compliant infrastructure on top of the UNICORE software. Grimme et al. [97] analyze the teikoku scheduling framework in the light of standards-compliance, and investigates the gaps between this framework and the proposed OGF Grid scheduling architecture.

5.2 Contributions to Interoperability

This thesis demonstrates how standard protocols and data formats contribute to middleware independent architectures for resource brokering, job management, and workflow enactment. A related contribution is the evaluation of proposed standards through an investigation of how these can be applied in practice. We describe some of the first results related to the usage of WSRF, JSDL, WS-Agreement, and the GLUE information model in software for Grid resource management. Furthermore, our proof-of-concept software tools provide easy integration with multiple middlewares.

Chapter 6

Design Considerations for Grid Software

This chapter discusses selected topics related to the development of Grid software, including the adoption of technologies associated with Service Oriented Architectures (SOAs). Also included is a description of the rationale behind the design choices taken during development of the software described in this thesis.

6.1 Service-Oriented Architectures

In a Grid middleware, it should be easy to add support for new types of resources such as sensors, instruments and alternative data sources. The construction of client tools is simplified if basic tasks such as allocation, invocation, notifications, and termination can be performed in a uniform manner, independent of the type of resource the client interacts with. The first generations of Grid software failed to fulfill these requirements, as illustrated by the early versions of the Globus toolkit that used resource type specific protocols, e.g., LDAP [60] for resource discovery, GridFTP [61] for data transfer, and GRAM [76] for job management. New resource types could only be integrated by the introduction of additional protocols, further adding to the protocol heterogeneity of the toolkit. More recently, many Grid projects have adopted principles from SOAs to overcome these limitations.

A SOA is "a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations" [156]. A key observation in SOAs is that a capability required by one entity (a person or an organization) often is offered by other entities. Needs and capabilities in a SOA are brought together by services. A service is described as the capability

to perform work for another, the specification of the capability, and the offer to perform the work [156]. All services in a SOA are described in a uniform manner through service interfaces. A service interface is a visibility mechanism that enables potential service consumers to decide if a particular service provider fulfills their requirements. Invocation of a service is transparent to both service location and the respective platforms of the service provider and the service consumer. With this transparency, services have the potential to increase reuse beyond that offered by software libraries as the latter are restricted to specific programming languages and/or platforms.

Figure 4 gives an overview of a SOA and illustrates the commonly applied publish-find-bind pattern. In the figure, a service provider publishes information about the capabilities offered by its service(s) in an index. A service consumer that requires some capability queries the index and locates a suitable service. In the last step, the service consumer binds to the service provider, and requests the capability.

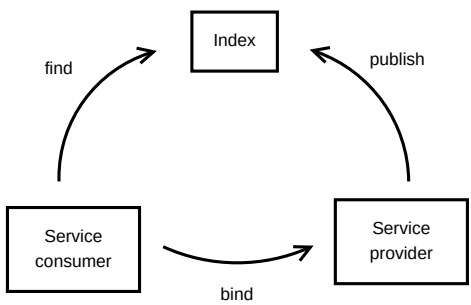


Figure 4: Publish-find-bind in a Service Oriented Architecture.

6.2 Web Services

Web services is a technology that can be used for realizing a SOA. The World Wide Web Consortium describes a Web service as a software system designed to support interoperable machine-to machine interaction over a network [214]. Web services are described using the Web Services Description Language (WSDL) [43] and communicate using SOAP [100], typically with HTTP as transport protocol. For Web services, an Universal Description, Discovery, and Integration (UDDI) [106] server can have the role of the index in the publish-find-bind scenario. Despite being a general architecture, Web services have some shortcomings when applied to Grid computing. Web services are stateless, i.e., there is no standard mechanism to store state information in a service between invocations and session management is thus complicated. Web services are also persistent and cannot be created and destroyed dynamically. Furthermore, the UDDI is designed for persistent services and can hence not handle [26] the

volatile nature of the Grid, where services may be available for limited periods of time only. The infeasibility of using pure Web services for Grid computing was the prime motivation for the design of the by now obsolete Open Grid Services Infrastructure (OGSI) [205]. The OGSI defines Grid services as an extension to the Web service concept. Grid services do, as opposed to Web services, have state and a limited lifetime. Lifetime management mechanisms allow Grid Services to be created and destroyed, the latter occurring either immediately or at a specified time. Furthermore, the OGSI specifies a mechanism for asynchronously notifying clients about changes in service state. An OGSI ServiceGroup uses soft-state registration of entries, making it more suitable than an UDDI server as an index for the Grid.

The OGSI specification did not gain widespread support, and was furthermore criticized by the Web services community, e.g., for being too tightly coupled to object orientation, for having poor support for existing Web service tools, and for being monolithic. To overcome these issues, OGSI was refactored into the Web Services Resource Framework (WSRF) [155] family of specifications that does not have the monolithic structure and many of the other shortcomings of OGSI. In WSRF, Web services are stateless and persistent. A stateless Web service can however have one or more stateful WS-Resources [94] associated with it. A client that invokes a WSRF-enabled service can specify, as part of the service address, which WS-Resource it wants to interact with. The WS-ResourceProperties [93] specification defines how state is stored in WS-Resources. WS-ResourceLifetime [192] defines mechanisms for lifecycle management of WS-Resources, but not of Web services as these are persistent in WSRF. The basic fault messages defined in the WS-Basefault specification [131] are used to increase consistency. In a WS-ServiceGroup [137], Web services and WS-Resources can be grouped together. Information registered in a ServiceGroup is, to avoid stale entries, removed unless it is renewed within a given time. The WS-BaseNotification [92] framework is not part of the WSRF family of specifications, but uses WSRF mechanisms to deliver asynchronous updates to interested entities and is often used together with the WSRF. An in-depth introduction to the areas of SOA and Web Services is beyond the scope of this thesis, but can be found in books [160, 190]. Joseph et al. [113] discuss the specifications for stateful Web Services and their impact on architectural principles of Grid computing.

6.3 Design Heuristics

It is of paramount importance for Grid system developers to study and understand the specific characteristics of Grid environments. These include the lack of total control over the considered Grid resources as well as the incomplete and typically outdated information that can be collected from them. Furthermore, failures occur at a regular basis, not only under rare conditions. One common misconception is that the Grid gives unlimited access to resources that can

be used freely if locally available ones become scarce. On the contrary, the demand for resources typically far exceeds the supply. Fundamental Grid middleware capabilities (as discussed in Section 1.3) help address heterogeneity in resource hardware and software. What must not be forgotten however is the more cumbersome heterogeneity in ownership and usage policies that is left to the developer of Grid software to handle. Resource availability is bound to vary over time, due both to faults and policy reasons. Failures in understanding the characteristics of Grid environments result in architectures and software tools that are unsuitable for Grid environments. Examples of this includes centralized software of omnipotent character that fail to function as expected due to competition from other, competing software tools.

The software tools constructed as part of this thesis are designed with great care to comply with the here discussed characteristics. Software is also built as we envision that it will work, without restricting ourselves to limitations of current environments. This includes support for functionalities that we foresee as inevitable for future systems. As an example, the advance reservation management system supports malleable reservations as this is foreseen as a critical functionality that most likely will be available in the future, although no contemporary batch schedulers support them. However, at the same time, care is taken to ensure that the tools are easily integrateable into existing infrastructures, and are fully operational also in environments that do not support the anticipated extensions. The software is not limited to use within a particular domain or application, but is rather indented as building blocks to simplify the construction of higher-level, domain-specific services.

Although there is large number of researchers worldwide that study Grid technologies and tools, surprisingly few have shared their ideas on topics such as design processes and software engineering for Grids. There is furthermore a focus on the application side, few contributions discuss the engineering of Grid middleware and other fundamental software. Abeti et al. [1] study how model driven architectures can be exploited in the design of SOAs. Hernández et al. [102] propose the use of domain-specific modeling to increase reuse when designing Grid applications. Experiences in using the CCA to build Grid applications are described by Gannon et al. [86]. In their discussion of the EveryWare system, Wolski et al. [220] describe a programming model and methodology for writing Grid programs and discuss some quality attributes. They also describe, from an application development perspective, how to use a *lingua franca* (mutually understood third language) to interface with the EveryWare system. Byun et al. [32] discuss adaptability, scalability, and reliability as well as demonstrate a WSRF-based resource provisioning framework with these characteristics.

With the increasing adoption of principles from SOAs by Grid projects, developers of Grid software would benefit from leveraging the software engineering work performed within the fields of SOAs [154, 194] and service-oriented computing [126, 161].

6.4 Contributions

This thesis demonstrates how principles of SOAs can be applied to Grid computing. This includes an investigation of how SOAs can be used to achieve loose coupling as demonstrated by the job management architecture in Paper IV, as well as improve reuse of workflow tools, the latter illustrated by papers V and VI. We investigate the characteristics of the future landscape of service-oriented Grid infrastructures in Paper VIII. This study also gives some of the first results regarding software engineering aspects of the design process for Grid middleware tools.

Chapter 7

Summary of the Papers

7.1 Paper I

Extending on our earlier work [67] in the area, Paper I focuses on algorithms and principles for the decentralized Grid brokering problem. A prediction of the Total Time to Delivery (TTD) for a Grid application is used as the main criteria in resource selection. The TTD includes the time required to perform the following operations: (i) transfer of the input files and executable to the selected Grid resource, (ii) wait for resource access, (iii) execute the application, and (iv) transfer of job output files to their requested location(s).

Paper I also discusses algorithms to estimate each part of the TTD, including the usage of bandwidth prediction tools for tasks (i) and (iv), the prediction of batch queue waiting times using either advance reservations or an estimate based on current resource load, and a benchmark-based mechanism to predict the application execution time. Paper I describes the implementation of resource selection algorithms based on prediction of the TTD in a proof-of-concept job submission tool for the NorduGrid/ARC middleware. Two approaches for advance reservations are also discussed and compared, one closely integrated with existing NorduGrid/ARC job submission mechanisms, the other a general service-based framework for reservation management.

7.2 Paper II

In Paper II, we investigate how principles of SOAs combined with standardization can be used to achieve adaptability in resource selection as well as Grid middleware independence. As a proof-of-concept, a general, service-based Grid resource brokering architecture is designed and implemented. This job submission service consists of a set of replaceable modules that each performs a well-defined task in the job submission process. The concept of replaceability is used also within some components, e.g., the resource selection algorithms

can be exchanged for alternative implementations.

The brokering architecture is designed to be as independent as possible of the Grid middleware used on the resources, and also to support any job description format on the client side. This interoperable design is achieved through the use of (proposed) standard formats and protocols, including WSRF, JSDL, GLUE, and WS-Agreement. These are used both internally in the job submission service and, if possible, in communication with resources. Interaction with a specific Grid middleware that uses non-standard data formats or protocols is handled through a set of well-defined integration points. The feasibility of this approach is demonstrated by the integration of the job submission service with the NorduGrid/ARC middleware.

Algorithmical contributions in Paper II include further development of the resource selection algorithms from Paper I, with focus on greater flexibility. Users may choose to select resources with the objective to achieve the earliest possible job completion, or the earliest possible job start. Furthermore, users may fine-tune the resource selection process by including a document with job preferences when invoking the broker. This document can be used to express job start time requirements and can also give helpful hints to the broker about the characteristics of the application.

7.3 Paper III

Paper III provides advances to the state-of-the-art in Grid resource coallocation, including the design, implementation, and analysis of an algorithm for arbitrarily coordinated allocations of resources. By viewing coallocation as a bipartite matching problem, we leverage well-known principles from graph theory to improve the likelihood of successfully allocating a set of resources. Paper III also contains a in-depth survey of existing coallocation approaches. By classifying contemporary coallocation algorithms and comparing these with methods from transaction management, we contribute to the general understanding of the coallocation problem.

The feasibility of the approach proposed in Paper II with a service-based architecture that enables interoperability through standardization is further demonstrated by the integration of the job submission service with Globus Toolkit 4 middleware in addition to the already supported NorduGrid/ARC. The implementation of integration plugins for a middleware is typically less than ten percent of the middleware-neutral code. This suggests that a feature-rich job submission tool for a Grid middleware can, with comparatively little effort, be obtained by implementing job submission service plugins for that middleware.

In Paper III, the performance of the job submission service is evaluated in depth, with focus on service response time and throughput for various Grid configurations. The coallocation algorithm is studied in a series of tests that both illustrate the performance dependencies of the different parts of the algorithm

and provide insight in the intrinsic complexity of the coallocation problem.

7.4 Paper IV

Paper IV investigates suitable architectural decompositions for job management software. The proof-of-concept Grid Job Management Framework (GJMF) further elaborates on ideas from papers II and III by extending the resource brokering software to also include job monitoring and control tools, as well as mechanisms that simplifies management of large sets of independent jobs. Another contribution is the study of fault tolerance techniques, through a multi-layer mechanism for automatic resubmission of failed jobs.

In the GJMF, the job management functionality is decomposed into a set of services that forms the following layers: the middleware abstraction layer that performs basic job management regardless of the Grid middleware at the resource side; the brokered job submission layer that offers resource discovery, resource selection, and job submission; and the reliable job submission layer that adds fault tolerance for single jobs or sets of jobs. Just as its predecessor, the GJMF makes use of the Web services, WSRF, and JSDL to promote standardization and increase interoperability. These aspects are further emphasized by implementing the draft resource selection services from OGSA [83] in the brokered job submission layer.

A performance evaluation demonstrates that the extra functionality offered by the GJMF services adds little overhead (down to 0.2 seconds per job). This evaluation suggests that for Grid application developers, multi-tiered job management frameworks such as the GJMF is an attractive alternative to interacting with the Grid middlewares directly.

7.5 Paper V

Paper V studies how Grid workflow capabilities can be decomposed into a loosely coupled architecture where capabilities are exposed as a set of workflow services with clear separation of concerns. By orchestrating these services, environments tailored to the needs of a certain group of users can be constructed with much less effort compared to (re)implementation of full-featured workflow systems. The feasibility of this approach is demonstrated by the design and implementation of the Grid Workflow Enactment Engine (GWEE). GWEE is not proposed as a replacement to existing workflow systems, but rather as a core component for developing new end-user tools and problem solving environments.

Paper V also demonstrates how workflow enactment can be completely decoupled from the processing of individual tasks, such as computational jobs or file transfers. It is further shown how Grid workflows can combine a data flow model of computation in which data dependencies decides the task execution order, with a control flow approach where control tokens are communicated to

initiate subsequent tasks. A contribution to workflow interoperability is the demonstration of how the GWEE tool can enact both data driven workflows expressed in the GWEE language and control driven workflows in the Karajan format.

7.6 Paper VI

Paper VI investigates how to define data flows that transparently integrate local and Grid workflows. In addition, the benefits of parameter sweep workflows are examined and a means for describing this type of workflow in an abstract and concise manner is introduced. Our parameter sweep mechanism extends the state-of-the-art in the area as it is not restricted to a predefined data distribution pattern. Both the problem of decoupling data flow from task specification and that of dynamically configuring workflow tasks during runtime are addressed by the introduction of a task template mechanism that delays the binding of task input parameters until workflow execution.

Paper VI further demonstrates the feasibility of the concept of a composable set of services with fundamental workflow capabilities by illustrating how a proof-of-concept client tool can be built on top of GWEE. With this client GUI, users can design, execute, and monitor workflows. The final contribution is a use case demonstrating how the developed mechanisms are employed to reduce the complexity of a particular bioinformatics problem - orthology detection analysis.

7.7 Paper VII

Paper VII is a comprehensive study of topics related to interoperability among scientific workflow systems. Rather than discussing if these workflow systems are completely interoperable or not at all, we argue that interoperability must be considered from three distinct dimensions: model of computation, workflow language, and workflow execution environment. Paper V illustrates workflow execution environment interoperability by showing how a workflow enactment engine can interoperate with multiple Grid middleware. Extending on this effort, the two most commonly used models of computation, Petri nets and dataflow networks, are studied with particular focus on how these have been adapted to fit the execution environment imposed by currently used Grid middleware.

Paper VII also investigates the minimum language constructs required for a language to be expressive enough to support scientific workflows. The fundamental differences in the respective execution environments of end-user desktop machines and the Grid suggest that whereas complex (Turing complete) languages could be used to describe locally executing workflows, simpler task coordination languages are preferred for the Grid. Leveraging on earlier results in the areas of theory of computation, compiler optimization, and visual

programming languages, Paper VII studies, from a workflow perspective, fundamental language constructs such as iterations, conditions, exceptions, and the implications of having a type system.

7.8 Paper VIII

With a starting point in the characteristics of Grid environments, Paper VIII describes the vision of a SOA-based ecosystem of Grid software components and outlines competitive factors for tools to "survive evolution" in such an environment. From these factors, the paper investigates design heuristics, design patterns, and quality attributes that are central to building software well adapted to the Grid ecosystem. These are divided into the following five groups. *Coexistence* includes aspects relating to decentralization, non-intrusiveness, and avoidance of resource over-consumption. Concerns for *composability* include well-defined interfaces, single-purpose components, and other characteristics that improve reuse. Ease of installation, configuration and use, as well as portability related concerns are all aspects of *Adoptability*. *Adaptability and changeability* affect ease of adaption into new or changed environments, and benefit from separation of policies from mechanisms. The last group is *interoperability* and the corresponding techniques to simplify ease of interaction with other software systems as well as the related standardization efforts. With a starting point in these five groups, the characteristics of software developed by the GIRD [201] team, including JSS (papers II and III), GJMF (Paper IV), GWEE (papers V, VI, and VII), as well as the SweGrid Accounting System (SGAS) [87], and FSGrid [66] are reviewed.

Chapter 8

Future Work

There are several potential directions of future work with a starting point in this thesis. These are grouped into three broad categories, the first one describing Grid brokering, job management, workflows, and SLA related topics. The last two sections of this chapter discuss anticipated extensions into the areas of virtualization and cloud computing.

8.1 Grid Resource Management

The current architecture for managing advance reservations is based on the WS-Agreement specification that defines an agreement request protocol message, with acceptance or rejection as the only possible answers. This makes multi-phase negotiation difficult, as experienced in the work with coallocation. A more general negotiation framework that includes offers and counter-offers, is described in WS-AgreementNegotiation [15], but this specification is not yet ready to be used. A full-featured negotiation protocol like WS-AgreementNegotiation would enable a resource broker to efficiently negotiate sophisticated agreements and hence meet a wider range of QoS requirements.

The coallocation algorithm introduced in Paper III can be improved, including development of better techniques to avoid (or resolve) conflicts that arise due to the competition for resources between the subjobs in a coallocated job. The construction of such algorithms would benefit from the development of theoretical models for the resource selection and coallocation processes. Further studies of the respective advantages and shortcomings of off-line and on-line coallocation algorithms should build on the large body of work on locking and optimistic concurrency control within the area of transactions. In addition to these algorithmic aspects, support for coordination of subjobs within a coallocated job is required. This should include mechanisms for subjobs to synchronize themselves prior to execution at their respective resources, and should leverage previous work in the area [28, 48, 50, 95].

Papers II and III demonstrate the feasibility of integrating the job submission service with the GT4 and NorduGrid/ARC Grid middlewares. A topic of current interest is the creation of a scheduling hierarchy, where higher-level Grid schedulers (brokers) negotiate with lower-level ones [118, 202].

If market-based Grids become increasingly more popular to the extent that they become the standard mechanism for Grid resource management, the resource selection algorithms described in this thesis can be adapted to select resources taking into account also Grid-economic parameters. This could e.g., include design and implementations of mechanisms that allow the user to specify the acceptable cost-performance tradeoff, as discussed in Section 2.1.

Future work in the workflow area includes investigating the interaction between workflow enactment and workflow scheduling. More specifically, the benefits and drawbacks of the respective approaches of preplanning and just-in-time scheduling, as well as the consequences of these for the design of enactment tools could be studied. Other future directions include the study of how streaming workflows [163] could extend the previous work on coallocation. Additional topics include further efforts in applications areas such as life sciences, e.g., by extending on the results in Paper VI.

8.2 Virtualization

The concept of logically dividing a physical computer into multiple, virtual machines has been used since the 1960's [3, 146]. The last few years, techniques such as paravirtualization [49] have improved the performance [229] of virtualization technologies and hence made adoption of these more feasible [180] for performance critical areas such as Grid computing. One motivating scenario for virtualization in Grids is the application-driven demand for tailored environments, with certain preinstalled software packages. Currently, one major obstacle in many production Grids is binary compatibility. It is cumbersome to use non-trivial (i.e., dynamically linked) applications that also may depend on external software, and be available for certain platforms only. One approach to overcome these problems is dynamic deployment, where software environments are installed and applications compiled as part of the job submission process. Although studied extensively [24] by, e.g., the OGF, the dynamic deployment approach has not been adopted to any larger extent. With virtualization techniques, dynamic deployment is greatly simplified as it can be performed by booting an already installed and configured virtual machine [117]. Another advantage with virtualization in Grid environments is the enhanced sandboxing functionality achieved through separation of (virtual) hardware.

Virtualization can also be used to provide checkpointing [191] and to enable migration of running jobs. Ubiquitous availability of an efficient checkpointing mechanism would fundamentally change Grid infrastructures and hence also affect the topics studied in this thesis. For example, resource brokering algorithms need not consider preemption of jobs upon runtime expiration, as

jobs can be restarted from a checkpoint. Furthermore, local batch scheduling algorithms can use aggressive backfilling and still avoid starvation of longer jobs, resulting in high utilization. Job migration mechanisms would enable job management tools to continuously monitor the performance of submitted jobs, and initiate migration upon performance declines. Another use case for checkpointing is improved fault tolerance, as failed jobs only need to restart from the last checkpoint. Checkpointing and migration mechanisms can also be used to improve Grid workflow management, e.g., by improving fault tolerance and shortening completion times for computational tasks.

8.3 Cloud Computing

Despite early enthusiasm, the Grid has yet not become the ubiquitous general purpose cyberinfrastructure used to access all types of resources. Being mostly focused on sharing of scientific instruments, high-performance computers and large-scale storage, existing Grid infrastructures do not easily meet the requirements of business applications and as a consequence, adoption outside academic environments is slow.

The currently popular vision for a general purpose infrastructure for providing IT capabilities as services goes under the name of cloud computing. Ideally, a cloud infrastructure should on demand adapt to changes in client request load by resizing itself. In order to achieve this adaptability, detailed and up to date monitoring information about server load and client performance metrics is required, as well as a (virtualization-powered) mechanism that enables the efficient migration and/or duplication of servers as needed. Notably, clients (service consumers) should ideally be unaware of the internal structure of the cloud and experience no degradation of QoS during server migration. Whereas the initial motivation for Grids was the interconnection of high-performance computers and expensive instruments mainly used in academic environments, the vision of a cloud as an infinitely scalable data center may better suited for industry needs.

Many problems, including security, accounting, and placement (resource selection) are similar in cloud and Grid environments. To become successful, research in cloud technologies should hence leverage the results obtained by the Grid community. There are however some notable differences between brokering of Grid jobs and provisioning of services in a cloud (i.e., service placement decisions). Services in a cloud may be hosted until further notice (potentially for a very long time), whereas Grid jobs typically have a known, finite duration. Performance metrics for service provisioning include minimization of SLA violations and costs (electricity, cooling etc.), and hence maximization of provider profit. Batch system scheduling (and also Grid brokering) on the other hand typically focus on response time, utilization, throughput, fairness, or a combination of these. These differences impose a series of research problems in how lessons learned from batch system oriented Grid brokering can be ap-

plied to service provisioning in time-shared cloud environments. The low-cost migration operation provided by virtualization technology justifies research on migration heuristics and cross-domain placement optimization.

8.3.1 RESERVOIR

The Resources and Services Virtualization without Barriers (RESERVOIR) project [173] is the venue for both ongoing work [224] and planned extensions to the topics studied in this thesis. The overall goal of RESERVOIR is to support service-oriented computing, in particular by dynamic provisioning of services as utilities. The prime motivation for the project is to avoid the costly over-provisioning currently used by data centers to ensure SLA compliance during peaks in demand. The RESERVOIR project will provide an open specification and a reference implementation of an infrastructure for federated clouds, where Grid and virtualization technologies along with business process management will enable efficient delivery of services. By resizing services on demand, potentially including migrating (parts of) them to other physical machines or even other, partnering data centers, a RESERVOIR infrastructure provider (a data center) will be able to optimize placement and hence minimize costs.

The design of heuristics for cost-aware SLA-compliant cross-site service placement share many similarities with the resource brokering scenarios investigated in this thesis. These similarities include complicating issues such as lack of control over remote sites, limited information about the load on these, and differences in usage policies. The existence of multiple brokers (placement decision modules) and competition among these for (potentially scarce) resources are other common characteristics of a federated cloud environment and a decentralized resource brokering scenario.

Bibliography

[1] L. Abeti, P. Ciancarini, and R. Moretti. Service oriented software engineering for modeling agents and services in grid systems. *Multiagent and Grid Systems*, 2(2):135–148, 2006.

[2] 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.

[3] R.J. Adair, R.U. Bayles, L.W. Comeau, and R.J. Creasy. A virtual machine system for the 360/40 - Cambridge scientific center report 320. Technical report, IBM, May 1966.

[4] 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.

[5] A.H. Alhusaini, V.K. Prasanna, and C.S. Raghavendra. A unified resource scheduling framework for heterogeneous computing environments. In V.K. Prasanna, editor, *8th Heterogeneous Computing Workshop, HCW'99*, pages 156–165, 1999.

[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] S. Ali, H.J. Siegel, M. Maheswaran, and D. Hensgen. Task execution time modeling for heterogeneous computing systems. In C. Raghavendra, editor, *Proceedings 9th Heterogeneous Computing Workshop (HCW 2000)*, pages 185–199, 2000.

[8] W.E. Allcock, I. Foster, and R. Madduri. Reliable data transport: A critical service for the Grid. Building Service Based Grids Workshop, GGF 11. Available at: www.globus.org/alliance/publications/papers/GGF11_RFTV-Final.pdf.

[9] 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):449–466, 2003.

[10] 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, 2205.

[11] G. Allen, T. Dramlitsch, I. Foster, N. Karonis, M. Ripeanu, E. Seidel, and B. Toonen. Supporting efficient execution in heterogeneous distributed computing environments with Cactus and Globus. In *Proceedings of Supercomputing 2001*, 2001.

[12] Amazon Elastic Compute Cloud (Amazon EC2). aws.amazon.com/ec2, June 2008.

[13] 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, August 2008.

[14] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web services agreement specification (WS-Agreement). https://forge.gridforum.org/sf/docman/do/downloadDocument/projects.graap-wg/docman.root.current_drafts/doc6090, November 2006.

[15] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web Services Agreement Negotiation Specification (WS-AgreementNegotiation). https://forge.gridforum.org/sf/go/doc6092?nav=1, November 2006.

[16] 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.136.pdf, August 2008.

[17] The Apache software foundation. The Apache ant project. http://ant.apache.org/, September 2008.

[18] C.G. Atkeson, A.W. Moore, and S. Schaal. Locally weighted learning. *Artificial Intelligence Review*, 11(1–5):11–73, 1997.

[19] F. Azzedin, M. Maheswaran, and N. Arnason. A synchronous co-allocation mechanism for Grid computing systems. *Cluster Computing*, 7(1):39–49, 2004.

[20] Emir M. Bahsi, Emrah Ceyhan, and Tevfik Kosar. Conditional workflow management: A survey and analysis. *Scientific Programming*, 15(4):283–297, 2007.

[21] X. Bai, L. Bölöni, D.C. Marinescu, H.J. Siegel, R.A. Daley, and I-J. Wang. A brokering framework for large-scale heterogeneous systems. In *Parallel and Distributed Processing Symposium, IPDPS 2006*, 2006.

[22] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC Storage Resource Broker. In S.A. MacKay and J.H. Johnson, editors, *Proceedings of CASCON'98*, 1998.

[23] J. Basney, M. Humphrey, and V. Welch. The MyProxy online credential repository. *Softw. Pract. Exper.*, 35(9):801–816, 2005.

[24] D. Bella, T. Kojo, P. Goldsack, S. Loughran, D. Milojicic, S. Schaefer, J. Tatemura, and P. Toft. Configuration description, deployment, and lifecycle management (CDDLM) foundation document. www.ogf.org/documents/GFD.50.pdf, August 2008.

[25] A. Bellouma, E. Deelman, and Z. Zhao (Eds.). Scientific workflows. *Scientific Programming*, 14(3–4), 2006.

[26] E. Benson, G. Wasson, and M. Humphrey. Evaluation of UDDI as a provider of resource discovery services for OGSA-based grids. In *Parallel and Distributed Processing Symposium, IPDPS 2006*, 2006.

[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] B. Bierbaum, C. Clauss, M. Pöppe1, S. Lankes, and Thomas Bemmerl. The new multidevice architecture of metampich in the context of other approaches to grid-enabled mpi. In B. Mohr, J. Larsson Träff, J. Worringen, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, LNCS 4192*, pages 184–193, 2006.

[29] S. Boag, D. Chamberlin, M.F. Fernández, D. Florescu, J. Robie, and J. Siméon (editors). XQuery 1.0: An XML Query Language. www.w3.org/TR/xquery, August 2008.

[30] 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.

[31] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP). http://tools.ietf.org/html/rfc2205, August 2008.

[32] E.-K. Byun and J.-S. Kim. Dynagrid: An adaptive, scalable, and reliable resource provisioning framework for WSRF-compliant applications. *J Grid Computing*, 2008. To appear, DOI:10.1007/s10723-008-9107-y.

[33] L.-C. Canon, E. Jeannot, R Sakellariou, and W. Zheng. Comparative evaluation of the robustness of dag scheduling heuristics. In S. Gorlatch, P. Fragopoulou, and T. Priol, editors, *Grid computing - achiements and prospects*, pages 63–74, 2008.

[34] H. Casanova and J. Dongarra. Netsolve: A network server for solving computational science problems. *Int. J. Supercomput. Appl.*, 11(3):212–223, 1997.

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

[36] C Castillo. *On the Design of Efficient Resource Allocation Mechanisms for Grids*. PhD thesis, North Carolina State University, 2008.

[37] C. Castillo, G.N. Rouskas, and K. Harfoush. On the design of online scheduling algorithms for advance reservations and qos in grids. In *2007 IEEE International Parallel and Distributed Processing Symposium*, 2007.

[38] CERN. CASTOR CERN Advanced STORage manager. http://www.cern.ch/castor, August 2008.

[39] A Chakrabarti, R. Dheepak, and S. Sengupta. Integration of scheduling and replication in data grids. In L. Bougé et al., editors, *High Performance Computing - HiPC 2004, LNCS 3296*, pages 375–385, 2004.

[40] C. Chapman, M. Musolesi, W. Emmerich, and C. Mascolo. Predictive resource scheduling in computational Grids. In *Parallel and Distributed Processing Symposium, IPDPS 2007*, 2007.

[41] K. Chard and K. Bubendorfer. A distributed economic meta-scheduler for the Grid. In T. Priol et al., editors, *CCGRID 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, pages 542–547, 2008.

[42] A.L. Chervenak, N. Palavalli, S. Bharathi, C. Kesselman, and R. Schwartzkopf. Performance and scalability of a replica location service. In *Proceedings of the International IEEE Symposium on High Performance Distributed Computing (HPDC-13)*, 2004.

[43] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl, June 2008.

[44] 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 Computat.: Pract. Exper.*, 18(10):1021–1037, 2006.

[45] J. Clark and S. DeRose (editors). XML Path Language (XPath) version 1.0. http://www.w3.org/TR/xpath, May 2008.

[46] The NextGRID consortium. NextGRID: Architecture for next generation grids. www.nextgrid.org, August 2008.

[47] CoreGRID. CoreGRID annual report 2007. http://www.coregrid.net/mambo/content/view/310/301/, May 2008.

[48] C. Coti, T. Herault, and S. Peyronnet. Grid services for MPI. In T. Priol et al., editors, *CCGRID 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, pages 417–424, 2008.

[49] S. Crosby and D. Brown. The virtualization reality. *ACM Queue*, pages 34–41, December/January 2006-2007.

[50] K. Czajkowski, I. Foster, and C. Kesselman. Resource co-allocation in computational Grids. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC-8)*, pages 219–228, 1999.

[51] J. Decker and J. Schneider. Heuristic scheduling of grid workflows supporting co-allocation and advance reservation. In B. Schulze et al., editors, *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*, pages 335–342. IEEE Computer Society, 2007.

[52] E. Deelman. Looking into the future of workflows: the challenges ahead. In I. Taylor et al., editors, *Workflows for e-Science*, pages 475–481. Springer-Verlag, 2007.

[53] 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*, 25(5):528–540, 2009.

[54] 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.

[55] E. Deelman and I. Taylor. Special issue on scientific workflows. *Journal of Grid computing*, 3(3–4):151–304, 2005.

[56] T. Delaitre, 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, 2005.

[57] T. Dierks and C. Allen. The TLS protocol version 1.0. http://www.ietf.org/rfc/rfc2246.txt, May 2008.

[58] J. Clark (editor). W3C XSL Transformations (XSLT) Version 1.0. www.w3.org/TR/xslt.html, August 2008.

[59] J. Treadwell (Editor). Open grid services architecture glossary of terms version 1.5 (gfd81). http://www.ogf.org/documents/GFD.81.pdf, May 2008.

[60] K. Zeilenga (editor). Lightweight Directory Access Protocol (LDAP): Technical specification road map. http://www.ietf.org/rfc/rfc4510.txt, November 2006.

[61] W. Allcock (editor). GridFTP: Protocol extensions to FTP for the Grid. http://www.ogf.org/documents/GFD.20.pdf, May 2008.

[62] EGEE. gLite - lightweight middleware for grid computing. www.cern.ch/glite, August 2008.

[63] EGEE. JRA1: workload management. http://egee-jra1-wm.mi.infn.it/egee-jra1-wm/, September 2008.

[64] A. Elghirani, R. Subrata, and A.Y. Zomaya. Intelligent scheduling and replication in datagrids: a synergistic approach. In B. Schulze et al., editors, *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*, pages 179–182, 2007.

[65] 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.

[66] 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.

[67] 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, LNCS 3732*, pages 1061–1070, 2006.

[68] European Telecommunications Standards Institute. ETSI GRID. http://portal.etsi.org/grid, August 2008.

[69] T. Fahringer, R. Prodan, R. Duan, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, and M. Wieczorek. ASKALON: A Grid Application Development and Computing Environment. In *6th International Workshop on Grid Computing*, pages 122–131. IEEE, 2005.

[70] 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.

[71] U. Farooq, S. Majumdar, and E. W. Parsons. Impact of laxity on scheduling with advance reservations in Grids. In *MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 319–324, Washington, DC, USA, 2005. IEEE Computer Society.

[72] International Organization for Standardization. ISO/IEC 2382-1 information technology - vocabulary - part 1: Fundamental terms, 1993.

[73] The Common Component Architecture Forum. The Common Component Architecture forum. http://www.cca-forum.org/, September 2008.

[74] I. Foster. What is the Grid? a three point checklist. www-fp.mcs.anl.gov/∼foster/Articles/WhatIsTheGrid.pdf, May 2008.

[75] 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.

[76] 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, 2006.

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

[78] I. Foster, J. Geisler, W. Nickless, W. Smith, and S. Tuecke. Software infrastructure for the I-WAY high performance distributed computing experiment. In *Proc. 5th IEEE Symposium on High Performance Distributed Computing*, pages 562–571, 1997.

[79] 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, August 2008.

[80] I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance reservations and co-allocation. In M. Zitterbart and G. Carle, editors, *7th International Workshop on Quality of Service*, pages 27–36. IEEE, 1999.

[81] 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.

[82] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *Int. J. Supercomput. Appl.*, 15(3), 2001.

[83] I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell, and J. Von Reich. The Open Grid Services Architecture, version 1.5, 2006. http://www.ogf.org/documents/GFD.80.pdf, October 2007.

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

[85] Geoffrey C. Fox and Dennis Gannon. Special issue: Workflow in grid systems. *Concurrency Computat.: Pract. Exper.*, 18(10):1009–1331, 2006.

[86] 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.

[87] 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.

[88] T. Glatard, J. Montagnat, D. Lingrand, and X. Pennec. Flexible and efficient workflow deployement of data-intensive applications on grids with MOTEUR. *Int. J. High Perf. Comput. Appl.*, 22(3):347–360, 2008.

[89] T. Glatard, G. Sipos, J. Montagnat, Z. Farkas, and P. Kacsuk. Workflow-level parametric study support by MOTEUR and the P-GRADE portal. In I. Taylor et al., editors, *Workflows for e-Science*, pages 279–299. Springer-Verlag, 2007.

[90] Globus. http://www.globus.org. February 2009.

[91] A. Goyeneche, G. Terstyanszky, T. Delaitre, and S. Winter. Improving Grid computing performance prediction using weighted templates. In S. Cox, editor, *Proceedings of the UK e-Science All Hands Meeting 2007*, pages 361–368, 2007.

[92] 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, June 2008.

[93] 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, June 2008.

[94] 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, June 2008.

[95] F. Gregoretti, G. Laccetti, A. Murli, G. Oliva, and U. Scafuri. MGF: A grid-enabled MPI library. *Future Generation Computer Systems*, 24(2):158–165, 2008.

[96] Grid Interoperability Now. http://wiki.nesc.ac.uk/read/gin-jobs. September 2006.

[97] C. Grimme, J. Lepping, A. Papaspyrou, P. Wieder, R. Yahyapour, A. Oleksiak, O. Wäldrich, and W. Ziegler. Towards a standards-based grid scheduling architecture. In S. Gorlatch, P. Fragopoulou, and T. Priol, editors, *Grid computing - achiements and prospects*, pages 147–158, 2008.

[98] A. S. Grimshaw and W. A. Wulf. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1):39–45, 1997.

[99] R. Gruber, V. Keller, P. Kuonen, M-C. Sawley, B. Schaeli, A. Tolou, M. Torruella, and T-M. Tran. Towards an intelligent Grid scheduling system. In R. Wyrzykowski, J. Dongarra, N. Meyer, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics, LNCS 3911*, pages 751–757. Springer Verlag, 2005.

[100] 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/, June 2008.

[101] 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.

[102] 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.

[103] T. Hey and A.E. Trefethen. The UK e-Science Core Programme and the Grid. *Future Generation Computer Systems*, 18(8):1017–1031, 2002.

[104] A. Hoheisel. User tools and languages for graph-based Grid workflows. *Concurrency Computat.: Pract. Exper.*, 18(10):1101–1113, 2006.

[105] R. Housley, W. Polk, W. Ford, and D. Solo. Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. http://www.ietf.org/rfc/rfc3280.txt, May 2008.

[106] http://uddi.xml.org/. UDDI. June 2008.

[107] http://www.sun.com/service/sungrid/SunGridUG.pdf. Sun^{TM} Grid compute utility - reference guide. August 2008.

[108] R. Huang, H. Casanova, and A.A. Chien. Using Virtual grids to simplify application scheduling. In *Parallel and Distributed Processing Symposium, IPDPS 2006*, 2006.

[109] E. Huedo, R.S. Montero, and I.M. Llorente. A framework for adaptive execution on Grids. *Softw. Pract. Exper.*, 34(7):631–651, 2004.

[110] Cluster Resources inc. Torque resource manager. http://www.clusterresources.com/pages/products/torque-resource-manager.php, August 2008.

[111] Institute of Electrical and Electronics Engineers. IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries., 1990.

[112] International Organization for Standardization. Information technology – database languages – SQL – part 1: Framework (SQL/framework). ISO/IEC 9075-1:2003.

[113] J. Joseph, M. Ernest, and C. Fellenstein. Evolution of grid computing architecture and grid adoption models. *IBM Systems journal*, 43(4), 2004.

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

[115] S. Kannan, P. Mayes, M. Roberts, D. Brelsford, and J. Skovira. *Workload Management with LoadLeveler*. IBM Corp., 2001.

[116] 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.

[117] K. Keahey, I. Foster, T. Freeman, and X. Zhang. Virtual workspaces: Achieving quality of service and quality of life in the Grid. *Scientific Programming*, 13(4):265–276, 2005.

[118] A. Kertesz and P. Kacsuk. Meta-broker for future generation grids: A new approach for a high-level interoperable resource management. In D. Talia et al., editors, *Proceedings of the CoreGRID Workshop on Grid Middleware*, 2007.

[119] A. Kertész, I. Rodero, and F. Guim. BPDL: A data model for grid resource broker capabilities. Technical report, CoreGRID, 2007. http://www.coregrid.net/mambo/images/stories/TechnicalReports/tr-0074.pdf, August 2008.

[120] G. Khanna, U. Catalyurek, T. Kurc, R. Kettimuthu, P. Sadayappan, and J. Saltz. A dynamic scheduling approach for coordinated wide-area data transfers using GridFTP. In *Parallel and Distributed Processing Symposium, IPDPS 2008*, 2008.

[121] T. Kosar and M. Livny. A framework for reliable and efficient data placement in distributed computing systems. *Journal of Parallel and Distributed Computing*, 65(10):1146–1157, 2005.

[122] 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.

[123] K. Kurowski, B. Ludwiczak, J. Nabrzyski, A. Oleksiak, and J. Pukacki. Dynamic grid scheduling with job migration and rescheduling in the GridLab resource management system. *Scientific Programming*, 12(4):263–273, 2004.

[124] H. Lamehamedi, B. Szymanski, Z. Shentu, and E. Deelman. Data replication strategies in grid environments. In *ICA3PP'02: Proceedings of the Fifth International Conference on Algorithms and Architectures for Parallel Processing*, pages 378–383, 2002.

[125] M. Lei, S.V. Vrbsky, and Z. Qi. Online Grid replication optimizers to improve system reliability. In *Parallel and Distributed Processing Symposium, IPDPS 2007*, 2007.

[126] P. Leong, C. Miao, and B-S. Lee. A survey of agent-oriented software engineering for service-oriented computing. *International Journal of Web Engineering and Technology*, 4(3):367–385, 2008.

[127] H. Li, J. Chen, Y. Tao, D. Groep, and L. Wolters. Improving a local learning technique for queue wait time predictions. In S.J. Turner et al., editors, *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2006)*, pages 335–342, 2006.

[128] H. Li, D. Groep, and L. Wolters. Efficient response time predictions by exploiting application and resource state similarities. In *6th International Workshop on Grid Computing (GRID 2005)*, pages 234–241, 2005.

[129] H. Li, D. Groep, L. Wolters, and J. Templon. Job failure analysis and its implications in a large-scale production grid. In *Second IEEE International Conference on e-Science and Grid Computing*. IEEE CS Press, 2006.

[130] A. Litke, D. Skoutas, K. Tserpes, and T. Varvarigou. Efficient task replication and management for adaptive fault tolerance in mobile Grid environments. *Future Generation Computer Systems*, 23(2):163–178, 2007.

[131] L. Liu and S. Meder (editors). Web Services Base Faults 1.2 (WS-BaseFaults). http://docs.oasis-open.org/wsrf/wsrf-ws_base_faults-1.2-spec-os.pdf, June 2008.

[132] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E.A. Leen, J. Tao, and Y. Zhao. Scientific workflow management and the Kepler system. *Concurrency Computat.: Pract. Exper.*, 18(10):1039–1065, 2006.

[133] B. Ludäscher and C. Goble. Special issue on scientific workflows. *SIGMOD Record*, 34(3), 2005.

[134] J. MacLaren. Advance reservations state of the art. http://www.fz-juelich.de/zam/RD/coop/ggf/graap/sched-graap-2.0.html, September 2006.

[135] J. MacLaren. HARC: the highly-available resource co-allocator". In *Proceedings of GADA'07, LNCS 4804*, pages 1385–1402, 2007.

[136] R.K. Madduri, C.S. Hood, and W.E. Allcock. Reliable file transfer in grid environments. In A. Jacobs, editor, *27th Annual IEEE Conference on Local Computer Networks (LCN)*, pages 737–738, 2002.

[137] T. Maguire, D. Snelling, and T. Banks (editors). Web Services Service Group 1.2 (WS-ServiceGroup). http://docs.oasis-open.org/wsrf/wsrf-ws_service_group-1.2-spec-os.pdf, June 2008.

[138] M.W. Margo, K. Yoshimoto, P. Kovatch, and P. Andrews. Impact of reservations on production job scheduling. In E. Frachtenberg and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 116–131. Springer, 2008. LNCS 4942.

[139] 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.

[140] C. Mateos, A. Zunino, and M. Campo. A survey on approaches to gridification. *Softw. Pract. Exper.*, 38(5):523–556, 2008.

[141] A. S. McGough, A. Afzal, J. Darlington, N. Furmento, A. Mayer, and L. Young. Making the Grid predictable through reservations and performance modelling. *The Computer Journal*, 48(3):358–368, 2005.

[142] A.S. McGough, W. Lee, J. Cohen, E. Katsiri, and J. Darlington. ICENI. In I. Taylor et al., editors, *Workflows for e-Science*, pages 395–415. Springer-Verlag, 2007.

[143] A.S. McGough and A. Savva. Implementation and interoperability experiences with the Job Submission Description Language(JSDL). http://www.ogf.org/Public_Comment_Docs/Documents/2008-07/draft-gwdejsdlexperience1.0007.pdf, August 2008.

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

[145] Message Passing Interface Forum. http://www.mpi-forum.org, June 2008.

[146] R.A. Meyer and L.H. Seawright. A virtual machine time-sharing system. *IBM Systems Journal*, 9(3):199–218, 1970.

[147] H. H. Mohamed and D. H. J. Epema. Experiences with the KOALA co-allocating scheduler in multiclusters. In *Proceedings of the International Symposium on Cluster Computing and the Grid (CCGRID2005)*, pages 784–791. IEEE Computer Society, 2005.

[148] 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.

[149] M. Morgan. ByteIO specification 1.0. http://www.ogf.org/documents/GFD.87.pdf, August 2008.

[150] M.M. Morgan and A.S. Grimshaw. Genesis II - standards based grid computing. In B. Schulze et al., editors, *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*, pages 611–618. IEEE Computer Society, 2007.

[151] M.A.S. Netto and R. Buyya. Rescheduling co-allocation requests based on flexible advance reservations and processor remapping. In *Proceedings of the 9th IEEE International Conference on Grid Computing, (Grid 2008)*, 2008.

[152] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. The Kerberos network authentication service (v5). http://www.ietf.org/rfc/rfc4120.txt, May 2008.

[153] S. Newhouse and A. Grimshaw. Independent software vendors (ISV) remote computing usage primer. http://www.ogf.org/Public_Comment_Docs/Documents/200807/ISV-V93.pdf, August 2008.

[154] E. Di Nitto, R.J. Hall, J. Han, Y. Han, A. Polini, K. Sandkuhl, and A. Zisman (editors). *The 2006 International Workshop on Service Oriented Software Engineering (IW-SOSE'06)*. ACM, 2006.

[155] OASIS. OASIS Web Services Resource Framework (WSRF) TC. http://www.oasis-open.org/committees/wsrf/, October 2007.

[156] OASIS Open. Reference Model for Service Oriented Architecture 1.0. http://www.oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf, June 2008.

[157] 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.

[158] 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 Computat.: Pract. Exper.*, 18(10):1067–1100, 2006.

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

[160] M.P. Papazoglou. *Web services: principles and technology*. Pearson Education Limited, 2008.

[161] M.P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-Oriented Computing: State of the art and research challenges. *Computer*, 40(11):38–45, 2007.

[162] S.G. Parker, D.M. Weinstein, and C.R. Johnson. The SCIRun computational steering software system. In E. Arge et al., editors, *Modern Software Tools in Scientific Computing*, pages 1–40. Birkhauser press, 1997.

[163] C. Pautasso and G. Alonso. Parallel computing patterns for grid workflows. In *Proc. of the HPDC2006 Workshop on Workflows in Support of Large-Scale Science (WORKS06) Paris France*, June 2006.

[164] M. Pereira, O. Tatebe, L. Luan, and T. Anderson. Resource namespace service specification. http://www.ogf.org/documents/GFD.101.pdf, August 2008.

[165] 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.

[166] M.L. Pinedo. *Scheduling — Theory, Algorithms, and Systems*. Springer Verlag, 2008.

[167] K. Plankensteiner, R. Prodan, T. Fahringer Attila Kertész, and P. Kacsuk. Fault-tolerant behavior in state-of-the-art Grid workflow management systems. Technical report, CoreGRID, 2007. http://www.coregrid.net/mambo/images/stories/TechnicalReports/tr-0091.pdf, August 2008.

[168] A. Pugliese, D. Talia, and R. Yahyapour. Modeling and supporting grid scheduling. *J Grid Computing*, 6(2):195–213, 2008.

[169] C. Qu. A Grid advance reservation framework for co-allocation and co-reservation across heterogeneous local resource management systems. In R. Wyrzykowski et al., editors, *PPAM 2007*, 2007.

[170] 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.

[171] H. Rajic, R. Borbst, W. Chan, F. Ferstl, J. Gardiner, A. Haas, B. Nitzberg, D. Templeton, J. Tollefsrud, and P. Tröger. Distributed resource management application API specification 1.0. http://www.ogf.org/documents/GFD.133.pdf, August 2008.

[172] K. Ranganathan and I. Foster. Simulation studies of computation and data scheduling algorithms for data grids. *Journal of Grid computing*, 1(1):53–62, 2003.

[173] Reservoir. Reservoir. http://www.reservoir-fp7.eu/, September 2008.

[174] M. Riedel and D. Mallmann. Standardization processes of the UNICORE Grid system. In J. Volkert et al., editors, *Proceedings of 1st Austrian Grid Symposium*, pages 191–203, 2005.

[175] T. Röblitz and A. Reinefeld. Co-reservation with the concept of virtual resources. In *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05)*, pages 398–406, 2005.

[176] 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.

[177] I. Rodero, F. Guim, J. Corbalan, L.L. Fong, Y.G. Liu, and S.M. Sadjadi. Looking for an evolution of Grid scheduling: Meta-brokering. In *Proceedings of the Second CoreGRID Workshop on Middleware at ISC2007 (CoreGRID-2007)*, 2007.

[178] H.G. Rotithor. Taxonomy of dynamic task scheduling schemes in distributed computing systems. *IEE Proceedings of Computers and Digital Techniques*, 141(1):1–10, 1994.

[179] D. De Roure, C. Goble, and R. Stevens. The design and realisation of the $^{\text{my}}$Experiment virtual research environment for social sharing of workflows. *Future Generation Computer Systems*, 2008. To appear, DOI:10.1016/j.future.2006.06.010.

[180] A.J. Rubio-Montero, E. Huedo, R.S. Montero, and I.M. Llorente. Management of virtual machines on globus Grids using GridWay. In *Parallel and Distributed Processing Symposium, IPDPS 2007*, 2007.

[181] 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.

[182] U. Schwiegelshohn, A. Tchernykh, and R. Yahyapour. Online scheduling in grids. In *Parallel and Distributed Processing Symposium, IPDPS 2008*, 2008.

[183] J. Seidel, O. Wäldrich, P. Wieder, Philipp, R. Yahyapour, and W. Ziegler. Using SLA for resource management and scheduling - a survey. In Domenico Talia, Ramin Yahyapour, and Wolfgang Ziegler, editors, *Grid Middleware and Services - Challenges and Solutions*, CoreGRID Series. Springer, 2008. Also published as CoreGRID Technical Report TR-0096.

[184] M. Siddiqui, A. Villazón, and T. Fahringer. Grid capacity planning with negotiation-based advance reservation for optimized QoS. In *the 2006 ACM/IEEE Conference on Supercomputing SC—06*, 2006.

[185] W. Smith. Improving resource selection and scheduling using predictions. In J. Nabrzyski, J.M. Schopf, and J. Węglarz, editors, *Grid Resource Management State of the art and future trends*, chapter 16. Kluwer Academic Publishers, 2004.

[186] W. Smith. Prediction services for distributed computing. In *Parallel and Distributed Processing Symposium, IPDPS 2007*, 2007.

[187] W. Smith, I. Foster, and V. Taylor. Using run-time predictions to estimate queue wait times and improve scheduler performance. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing, LNCS 1459*, pages 202–219, 1999.

[188] W. Smith, I. Foster, and V. Taylor. Scheduling with advance reservations. In *14th International Parallel and Distributed Processing Symposium*, pages 127–132, 2000.

[189] Q. Snell, M. Clement, D. Jackson, and C. Gregory. The performance impact of advance reservation meta-scheduling. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing: IPDPS 2000 Workshop, JSSPP 2000, LNCS 1911*, pages 137–153. Springer Berlin / Heidelberg, 2000.

[190] B. Sotomayor and L. Childers. *Globus Toolkit 4 - Programming Java Services*. Elsevier, 2006.

[191] B. Sotomayor, K. Keahey, and I. Foster. Combining batch execution and leasing using virtual machines. In *HPDC - The ACM/IEEE International Symposium on High Performance Distributed Computing*, 2008.

[192] L. Srinivasan and T. Banks (editors). Web Services Resource Lifetime 1.2 (WS-ResourceLifetime). http://docs.oasis-open.org/wsrf/wsrf-ws_resource_lifetime-1.2-spec-os.pdf, June 2008.

[193] H. Stockinger. Defining the grid: a snapshot on the current view. *J Supercomput*, 42(1):3–17, 2007.

[194] Z. Stojanovic and A. Dahanayake. *Service-oriented Software System Engineering: Challenges and Practices*. Idea Group Inc, 2005.

[195] 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.

[196] I. Taylor, E. Deelman, D. Gannon, and M. Shields. *Workflows for e-Science*. Springer-Verlag, 2007.

[197] I. Taylor, M. Shields, I. Wang, and A. Harrison. The Triana workflow environment: architecture and applications. In I. Taylor et al., editors, *Workflows for e-Science*, pages 320–339. Springer-Verlag, 2007.

[198] I. Taylor, M. Shields, I. Wang, and O. Rana. Triana applications within grid computing and peer to peer environments. *J Grid Computing*, 1(2):199–217, 2003.

[199] J. Taylor. News from the e-Science programme. http://www.rcuk.ac.uk/escience/news/firstphase.htm, July 2008.

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

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

[202] N. Tonellotto, R. Yahyapour, and Ph. Wieder. A proposal for a generic Grid scheduling architecture. Technical report, CoreGRID, 2006. http://www.coregrid.net/mambo/images/stories/TechnicalReports/tr-0015.pdf, August 2008.

[203] H. Topcuoglu, S. Hariri, and W. Min-You. Task scheduling algorithms for heterogeneous processors. In V.K. Prasanna, editor, *Heterogeneous Computing Workshop, 1999. (HCW '99) Proceedings*, pages 3–14, 1999.

[204] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Modeling user runtime estimates. In *The 11th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), LNCS 3834*, pages 1–35, 2005.

[205] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maguire, T. Sandholm, P. Vanderbilt, and D. Snelling. Open Grid Services Infrastructure (OGSI) version 1.0, Global Grid Forum Draft Recommendation. http://www.globus.org/alliance/publications/papers/Final_OGSI_Specification_V1.0.pdf, June 2008.

[206] S. Tuecke, V. Welch, D. Engert, L. Pearlman, and M. Thompson. Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile. http://www.ietf.org/rfc/rfc3820.txt, May 2008.

[207] S. Ullah and I. Ahmad. Non-cooperative, semi-cooperative, and cooperative games-based grid resource allocation. In *Parallel and Distributed Processing Symposium, IPDPS 2006*, 2006.

[208] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[209] J.L. Vázquez-Poletti, E. Huedo, R.S. Montero, and I.M. Llorente. A comparison between two grid scheduling philosophies: EGEE WMS and GridWay. *Multiagent and Grid Systems*, 3(4):429–439, 2007.

[210] S. Venugopal, X. Chu, and R. Buyya. A negotiation mechanism for advance resource reservations using the alternate offers protocol. In *Quality of Service, 2008. IWQoS 2008. 16th International Workshop on*, pages 40–49, 2008.

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

[212] J. Vollbrecht, P. Calhoun, S. Farrell, L. Gommans, G. Gross, B. de Bruijn, C. de Laat, M. Holdrege, and D. Spence. AAA authorization framework. http://www.ietf.org/rfc/rfc2904.txt, August 2008.

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

[214] W3C. Web Services Activity. http://www.w3.org/2002/ws/, June 2008.

[215] O. Wäldrich, P. Wieder, and W. Ziegler. A meta-scheduling service for co-allocating arbitrary types of resources. In R. Wyrzykowski, J. Dongarra, N. Meyer, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics, LNCS 3911*, pages 782–791. Springer Verlag, 2005.

[216] J. Wang, L-Y. Zhang, and Y-B. Han. Client-centric adaptive scheduling for service-oriented applications. *J. Comput. Sci. and Technol.*, 21(4):537–546, 2006.

[217] V. Welch. Grid Security Infrastructure message specification. http://www.ogf.org/documents/GFD.78.pdf, May 2008.

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

[219] M. Wieczorek, R. Prodan, and T. Fahringer. Scheduling of scientific workflows in the ASKALON grid environment. *SIGMOD Record*, 34(3):56–62, 2005.

[220] R. Wolski, J. Brevik, G. Obertelli, N. Spring, and A. Su. Writing programs that run EveryWare on the computational Grid. *IEEE transactions on parallel and distributed systems*, 12(10), 2001.

[221] R. Wolski, J. Brevik, J. S. Plank, and T. Bryan. Grid resource allocation and control using computational economies. In F. Berman, G. Fox, and A. Hey, editors, *Grid Computing: Making The Global Infrastructure a Reality*, chapter 32. John Wiley & Sons, 2003.

[222] M.Q. Xu. Effective metacomputing using LSF multicluster. In *Cluster Computing and the Grid, 2001*, pages 100–105, 2001.

[223] R. Yahyapour. Considerations for resource brokerage and scheduling in Grids. In G. R. Joubert, W. E. Nagel, F. J. Peters, and W. V. Walter, editors, *Parallel Computing: Software Technology, Algorithms, Architectures and Applications, PARCO 2003, Dresden, Germany*, pages 627–634, 2004.

[224] M. Ben Yehuda, O. Biran, D. Breitgand, K. Meth, B. Rochwerger, E. Salant, E. Silvera, S. Tal, Y. Wolfsthal, J. C'aceres, J. Hierro, W. Emmerich, A. Galis, L. Edblom, E. Elmroth, D. Henriksson, F. Hernández, J. Tordsson, A. Hohl, E. Levy, A. Sampaio, B. Scheuermann, M. Wusthoff, J. Latanicki, G. Lopez, J. Marin-Frisonroche, A. Dörr, F. Ferstl, S. Beco, F. Pacini, I. Llorente, R. Montero, E. Huedo, P. Massonet, S. Naqvi, G. Dallons, M. Pezzé, A. Puliato, C. Ragusa, M. Scarpa, and S. Muscella. RESERVOIR - an ICT infrastructure for reliable and effective delivery of services as utilities. Technical report, IBM Haifa Research Laboratory, 2008.

[225] C.S. Yeo and R. Buyya. A taxonomy of market-based resource management systems for utility-driven cluster computing. *Softw. Pract. Exper.*, 36(13):1381–1419, 2006.

[226] B. Plale Y.L. Simmhan and D. Gannon. A survey of data provenance in e-Science. *SIGMOD Record*, 34(3):31–36, 2005.

[227] K. Yoshimoto, P. Kovatch, and P. Andrews. Co-scheduling with user-settable reservations. In *IEEE Mass Storage Conference*, 2005.

[228] 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.

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

[230] J. Yu and R. Buyya. A taxonomy of workflow management systems for Grid computing. *J. Grid Computing*, 3(3–4):171–200, 2006.

I

Paper I

Grid Resource Brokering Algorithms Enabling Advance Reservations and Resource Selection Based on Performance Prediction*

E. Elmroth and J. Tordsson

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

Abstract: We present algorithms, methods, and software for a Grid resource manager, responsible for resource brokering and scheduling in production Grids. The broker selects computing resources based on actual job requirements and a number of criteria identifying the available resources, with the aim to minimize the total time to delivery for the individual application. The total time to delivery includes the time for program execution, batch queue waiting, input/output data transfer, and executable staging. The main features of the resource manager include advance reservations, resource selection based on computer benchmark results and network performance predictions, and a basic adaptation facility. The broker is implemented as a built-in component of a job submission client for the NorduGrid/ARC middleware.

Key words: Resource broker, Grid scheduling, Benchmark-based performance prediction, Advance reservations.

* By permission of Elsevier B.V.

Available online at www.sciencedirect.com
ScienceDirect
Future Generation Computer Systems 24 (2008) 585–593
FGCS FUTURE GENERATION COMPUTER SYSTEMS
www.elsevier.com/locate/fgcs

Grid resource brokering algorithms enabling advance reservations and resource selection based on performance predictions☆

Erik Elmroth, Johan Tordsson

Department of Computing Science, Umeå University, SE-901 87 Umeå, Sweden
HPC2N, Umeå University, SE-901 87 Umeå, Sweden

Received 19 January 2007; accepted 3 June 2007
Available online 13 June 2007

Abstract

We present algorithms, methods, and software for a Grid resource manager, that performs resource brokering and job scheduling in production Grids. This decentralized broker selects computational resources based on actual job requirements, job characteristics, and information provided by the resources, with the aim to minimize the total time to delivery for the individual application. The total time to delivery includes the time for program execution, batch queue waiting, and transfer of executable and input/output data to and from the resource. The main features of the resource broker include two alternative approaches to advance reservations, resource selection algorithms based on computer benchmark results and network performance predictions, and a basic adaptation facility. The broker is implemented as a built-in component of a job submission client for the NorduGrid/ARC middleware.

Keywords: Resource broker; Grid scheduling; Runtime predictions; Performance-based resource selection; Advance reservations

1. Introduction

The task of a Grid resource broker and scheduler is to dynamically identify and characterize the available resources, and to select and allocate the most appropriate resources for a given job. The resources are typically heterogeneous, locally administered, and accessible under different local policies. A decentralized broker, as the one considered here, operates without global control, and its decisions are entirely based on the information made available by individual resources and index servers providing lists of available resources and aggregated resource information. For an introduction to typical resource brokering requirements and solutions, see [1–3].

The overall objective of the broker presented here is to select the resource that gives the shortest time for job completion for each job, including the time for file staging, batch queue waiting, and job execution. In order to perform this selection, the broker needs to predict the times required to perform each of these tasks for all resources considered, predictions that can be rather difficult to make due to the heterogeneity and the dynamic nature of the Grid.

The performance differences between Grid resources and the fact that their relative performance characteristics may vary for different applications makes predictions of job execution time difficult; see, e.g., [4–6]. We address this problem with a benchmark-based procedure for execution time prediction. Based on the user's identification of relevant benchmarks and an estimated execution time on some specified resource, the broker estimates the execution time for all resources of interest. This requires that a relevant set of benchmark results are available from the resources' information systems. Notably, the results do not necessarily have to be for standard computer benchmarks only. On the contrary, performance results for real application codes for some test problem are often to recommend. The time prediction for the staging of executable,

☆ This work is a revised and extended version of: E. Elmroth, J. Tordsson, A Grid resource broker supporting advance reservations and benchmark-based resource selection, in: J. Dongarra, K. Madsen, J. Wasniewski (Eds.), State-of-the-art in Scientific Computing, in: LNCS, vol. 3732, Springer-Verlag, 2006, pp. 1061–1070. It has been funded by The Swedish Research Council (VR) under contracts 343-2003-953 and 621-2005-3667, and it has been conducted using the resources of High Performance Computing Center North (HPC2N).

E-mail addresses: elmroth@cs.umu.se (E. Elmroth), tordsson@cs.umu.se (J. Tordsson).

0167-739X/$ - see front matter © 2008 Published by Elsevier B.V.
doi:10.1016/j.future.2007.06.001

input and output files are based on network performance predictions and file size information.

An advance reservation capability in the resource broker is vital for meeting deadlines for time critical applications and for enabling co-allocation of resources in highly utilized Grids. A reservation feature also provides a guaranteed alternative to predicting batch queue waiting times. Such a feature naturally depends on the reservation support provided by local schedulers [7] and the use of advance reservations also has implications on the utilization of each local resource [8]. For general discussions about resource reservations (and co-allocation), see, e.g., [9,10].

Our resource brokering software is mainly intended for the NorduGrid [11] and SweGrid [12] infrastructures. These are both production environments for 24 hour per day Grid usage, built on the Globus Toolkit 2-based NorduGrid/ARC (in the following denoted ARC, which is an abbreviation for Advanced Resource Connector) middleware [15]. The broker is implemented as a built-in component of an ARC job submission client. For an extension of this work towards a service-oriented framework, see [13,14].

The outline of the paper is as follows. Section 2 gives a brief introduction to the ARC software and the general resource brokering problem. The main algorithms and techniques of our resource broker are presented in Section 3. Minor extensions to the ARC user interface are presented in Section 4. Sections 5 and 6 present future work, some concluding remarks and acknowledgements, respectively, followed by a list of references.

2. Background and motivation

Our development of resource brokering algorithms and prototype implementations is focused mainly on the infrastructure and usage scenarios typical for NorduGrid and SweGrid. The main Grid middleware used is the ARC. The Grid resources are typically Linux-based clusters (in the rest of this paper, the word cluster is often used instead of the more general term Grid resource). NorduGrid includes over 50 clusters, totally comprising over 5000 CPUs, distributed over 13 countries with most of the clusters located in the Nordic countries. SweGrid currently consists of six Swedish clusters, each with 100 CPUs.

2.1. The ARC software

The ARC middleware is based on (de facto) standard protocols and software such as OpenLDAP [16], OpenSSL [17] and Globus Toolkit version 2 [18,19]. The latter is not used in full, as some Globus components such as the GRAM (with the gatekeeper and jobmanager) are replaced by custom components [15]. Moreover, the Globus GSI (Grid Security Infrastructure) is used, e.g., for certificate and proxy management.

The ARC user interface consists of command line tools for job management. Users can submit jobs, monitor the execution of their jobs and cancel jobs. The resource broker is an integral part of the job submission tool, `ngsub`. Other tools allow, for example, the user to retrieve results from jobs, obtain a preview of job output and remove all files generated by the job from the remote resource. Communication with remote resources is handled by a GridFTP client module.

Each Grid resource runs an ARC *GridFTP server*. When submitting a job, the user invokes the broker that uploads an xRSL job request to the GridFTP server on the selected resource. The xRSL is an extended subset of the Resource Specification Language, originally proposed by the Globus project. The ARC GridFTP server specifies plug-ins for custom handling of FTP protocol messages. In ARC, these are used for Grid access to the local file systems of the resources, to handle Grid access control lists and, most important, for management of Grid jobs.

Each resource also runs a *Grid manager* that manages the Grid jobs through the various phases of their execution. The Grid manager periodically searches for new jobs accepted by the GridFTP server. For each new job, the xRSL job description is analyzed, and any required input files are staged to the resource. Then, the job description is translated into the language of the local scheduler, and the job is submitted to the batch system. Upon job completion, the Grid manager stages the output files to the location(s) specified in the job description. The Grid manager can be configured, at each job state change, to execute a script that intercepts (and possibly cancels) the job. These scripts form natural plug-in points for, for example, accounting [20], and, as demonstrated in Section 3.2.1, for the authorization of jobs that request advance reservations.

2.2. The resource brokering problem

One way to classify a Grid resource broker is by the scope of its operations. A centralized broker manages and schedules all jobs submitted to the Grid, whereas a decentralized broker typically handles jobs submitted by a single user only. Centralized brokers have a good knowledge and control of the jobs and resources and can hence produce good schedules, but such a broker can easily become a performance bottleneck and a single point of failure. A decentralized brokering architecture, on the other hand, scales well and makes the Grid more fault-tolerant, but the incomplete information available to each instance of the broker makes the scheduling problem more difficult.

We can distinguish between two major categories of the scheduling policies used by Grid resource brokers. System-oriented brokers often strive to maximize overall system throughput, average response times, fairness, or a combination of these, whereas user-oriented scheduling systems try to optimize the performance for an individual user, typically by minimizing the response time for each job submitted by that user. This is done regardless of the impact on the overall scheduling performance and at the expense of competing schedulers. A centralized broker typically performs system-oriented scheduling whereas a decentralized broker often uses a user-oriented policy. Grid systems utilizing a centralized broker include, e.g., EMPEROR (user-oriented) [21] and Condor (system-oriented) [22]. Distributed brokers are implemented

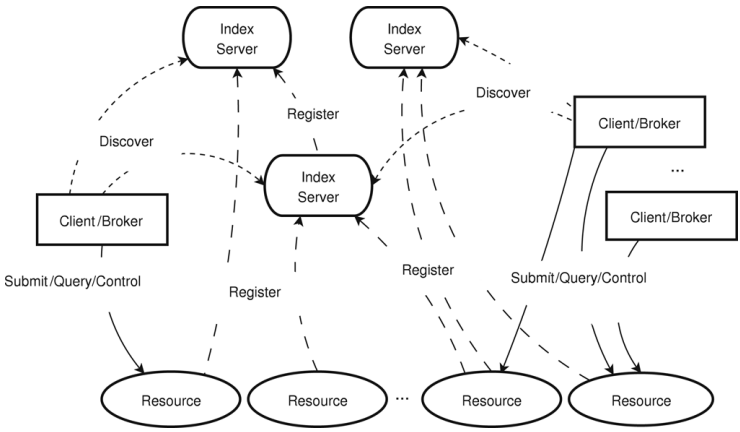


Fig. 1. Interactions between resources, index servers and brokers.

by, e.g., Nimrod-G (user-oriented) [23] and GridWay (user-oriented) [24]; the latter can also be deployed as a centralized broker. For further discussions on classifications of brokers, see, e.g., [25,26].

Fig. 1 illustrates a Grid with decentralized resource brokers. Each Grid resource registers itself to one or more index servers, which in turn can register to higher level index servers, thus forming an index server hierarchy. All clients accessing the Grid resources use their own brokers. Each broker, optionally, contacts one or more index servers to discover what Grid resources are available. The brokers query individual clusters for detailed resource information and perform job submission and job control by communicating directly with the resources.

In the following, we focus on algorithms and software for a decentralized user-oriented resource broker. Our broker seeks to fulfill the user's resource requests by selecting the resources that best suit the user's application. In this context, selecting the most suitable resources means identifying the resources that provide the shortest Total Time to Delivery (TTD) for the job. The TTD is the total time elapsed from the user's job submission until the output files are stored where requested. This includes the time required for transferring input files and executable to the resource, the waiting time, e.g., in a batch queue, the actual execution time, and the time to transfer the output files to the requested location(s).

3. Resource brokering algorithms

Our main brokering algorithm performs a series of tasks, e.g., it processes the xRSL specifications in the job requests, discovers and characterizes the resources available, estimates the TTD for each resource of interest, makes advance reservation of resources, and performs the actual job submission. Algorithm 1 presents a high-level outline of the tasks performed.

The input xRSL specification(s) contain one or more job requests including information about the application to run (e.g., executable, arguments, input/output files), actual job requirements (e.g., amount of memory needed, architecture requirements, execution time required), and optionally, job characteristics that can be used to improve the resource selection (e.g., listing of benchmarks with performance characteristics relevant for the application). The broker input can also include a request for advance reservations.

In Step 1 of Algorithm 1, the user's request is processed and split into individual job requests. In Step 2, the broker discovers what resources are available by contacting one or more index servers. The specific characteristics of the resources found are identified in Step 3, by querying each individual resource. Each resource may provide static information about architecture type, memory configuration, CPU clock frequency, operating system, local scheduling system, etc., and dynamic information about current load, batch queue status and various usage policies. Steps 2 and 3 are both performed by LDAP queries sent from the broker to the index servers and the resources, respectively. The actual brokering process is mainly performed in Step 4, which is repeated for each job request. In Step 5, resources are evaluated according to the requirements in the job request and only the appropiate resources are kept for further investigation. Step 6 predicts the performance of each resource by estimating the TTD, a step that may include the creation of advance reservations. Then, the currently considered job is submitted in the loop started at Step 7. The loop is repeated until either the job is successfully submitted or all submission attempts fail, the latter causing the job to fail. In Step 8, the best of the (remaining) clusters is selected for submission. In Step 9, the actual job submission is performed. Steps 10 and 11 test if the submission fails, and, if so, the cluster is discarded and the algorithm retries from Step 7. In Step 14, after the submission of one job is completed, any non-utilized reservations are released. Finally, in Step 16, job identifiers for all successfully submitted jobs are returned. These job identifiers are obtained from successful execution of Step 9.

Algorithm 1 Job submission and resource brokering.

Require: xRSL-specification(s) of one or more job requests.
Ensure: Returns job identifier(s) for the submitted job(s).

```
1:  Validate the xRSL specification(s) and create a list of all individual job requests.
2:  Contact one or more index servers to obtain a list of available clusters.
3:  Query each resource for static and dynamic resource information (hardware and software characteristics, current status and
    load, etc).
4:  for each job do
5:     Filter out clusters that do not fulfill the job requirements on memory, disk space, architecture, etc., and clusters that the user
       is not authorized to use.
6:     Estimate TTD for each remaining resource (see Section 3.1). If requested, advance reservations are created during this
       process.
7:     repeat
8:        Select the (remaining) cluster with the shortest predicted TTD.
9:        Submit the job to the selected resource.
10:       if submission fails then
11:          Discard the cluster.
12:       end if
13:    until the job is submitted or no appropriate clusters are left to try.
14:    Release any reservations made to non-selected clusters.
15: end for
16: Return the job identifier(s).
```

Notably, Algorithm 1 does not reorder the individual job requests internally (when multiple jobs are submitted in a single invocation). This can possibly be done in order to reduce the average batch queue waiting time, at least by submitting shorter jobs before longer ones given that they require the same number of CPUs. However, in the general case, factors such as local scheduling algorithms (backfilling) and competing users make the advantage of job reordering less obvious.

Fig. 2 presents a sequence diagram for the tasks performed in Algorithm 1. The interactions are between the broker on the client machine, the Grid resources, and an index server, each typically running on a separate host. The broker's different interactions with a resource are performed with three different components, for requesting resource information (the index server), creating an advance reservation (any of the reservation components presented in Section 3.2), and submitting a job (the GridFTP server). Note that the steps for requesting resource information, and possibly also for requesting an advance reservation, are normally performed for a number of resources before one is selected for the final job submission. In the sequence diagram in Fig. 2, the GetInformation and CreateReservation operations are invoked for all resources of interest, whereas the SubmitJob request is sent to the selected resource only.

In the following presentation, we focus on the more intricate details of performing advance reservations and the algorithms used to predict the TTD.

3.1. Estimating the total time to delivery

The prediction of the TTD, from the user's job submission to the final delivery of output files to the requested storage location(s) requires that the time to perform the following operations is estimated:

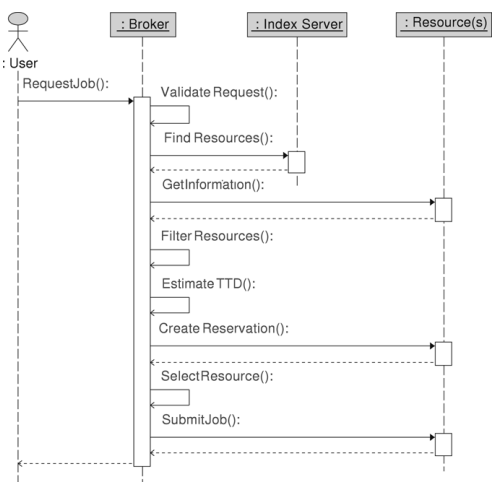


Fig. 2. Sequence diagram for job submission.

(i) Stage in: transfer of input files and executable to the resource,
(ii) Waiting, e.g., in a batch queue and for operation (i) to complete,
(iii) Execution, and,
(iv) Stage out: transfer of output files to the requested location(s).

Notably, the waiting time is here defined as the maximum of the time for stage in and all other waiting times before the

job can actually start to execute. The estimated TTD is given as the sum of the estimated times for operations (ii), (iii) and (iv). If the time for stage out cannot be estimated due to lack of information about output file sizes, that part is simply omitted from the TTD. Below, we summarize how we make these estimates.

3.1.1. Benchmark-based execution time predictions

The execution time estimate needs to be based both on the performance of the resource and the characteristics of the application, as the relative performance difference between different computing resources typically varies with the character of the application. In order to do this, we give the user the opportunity to specify one or more benchmarks with performance characteristics similar to those of the application. This information is given together with an execution time estimate on a resource with a specified benchmark result.

We remark that what is here referred to as benchmarks do not exclusively have to be standard computer benchmarks. On the contrary, for applications commonly used on resources, the use of real application codes is recommended for this benchmarking. For example, in Swegrid, benchmarking with the ATLAS software [27] would be relevant for a large group of high-energy physics users. An alternative to running the full application codes for benchmarking is to run small test programs representative for the application [28].

To run the benchmarking code at the time of each job submission would impose a significant overhead as one additional job (the test code) would have to be submitted, and to wait in the batch queue on each resource of interest, before submission of the real job. Therefore, our approach is to run the benchmarks (e.g. test codes) once for each resource and then have the results published in the information system. As the current usage of Grid resources typically involves a very large number of jobs requiring a small number of applications, this minor extra work, e.g., at the time of the software installation, is well motivated. In addition, a few standard benchmarks can be included for general usage, as is currently done in NorduGrid.

Based on the benchmark results published by each individual resource and the user's specification of relevant benchmarks for the application, the broker makes execution time estimates for all resources of interest. In doing this, we assume linear scaling of the application in relation to the benchmark, i.e. a resource with a benchmark result a factor k better is assumed to execute the application a factor k faster.

The user can specify n benchmarks as triples $\{b_i, r_i, t_i\}$, $i = 1, \ldots, n$, where b_i is the benchmark name and r_i is the benchmark result on a system where the application requires the time t_i to execute. The broker matches these specifications with the benchmark results provided by each cluster. For each requested benchmark that is available for a resource, an execution time for the application is predicted using linear scaling of the benchmark result.

If the cluster provides results for some, but not for all requested benchmarks, the broker compensates for the uncertainty by taking the corresponding time estimates for the missing benchmark(s) to be a penalty factor c times the longest execution time estimated from other benchmarks for that cluster. The default penalty factor is $c = 1.25$, but this can be reconfigured by the user.

The predicted execution time is used twice: as part of the TTD comparison during resource selection and as the requested execution time at job submission. For the TTD comparison, the average of the n execution time estimates is used as it reflects the overall resource performance with respect to the user-specified benchmarks. The maximum of the n predicted values is used as the requested execution time. This gives an accurate, yet conservative estimate of the execution time. We remark that a sufficient but short execution time estimate may lead to an earlier job start due to standard batch system scheduling algorithms. A good estimate is also more likely to be not too short, and hence reduces the risk of job cancellation by the local scheduler.

3.1.2. File transfer time predictions

The time estimation for the stage in and stage out procedures are based on the actual (known) sizes of input files and the executable file, user-provided estimates for the sizes of the output files, and network bandwidth predictions. If any of the input files are replicated, time estimations are made for each copy of the file. The current version of the ARC Grid manager does not, however, use these estimates when selecting which copy of a replicated file to stage to the resource prior to execution, but, rather, chooses a random replica.

The network bandwidth predictions are performed using the Network Weather Service (NWS) [29]. NWS combines periodic bandwidth measurements with statistical methods to make short-term predictions for the available bandwidth.

3.1.3. Waiting time estimations

The most accurate method to predict the waiting time is to create an advance reservation for the job, which gives a guaranteed start time rather than an estimate. However, a reservation-based approach to predicting the waiting time cannot be used if the resource lacks support for advance reservations (via one of the mechanisms described in Section 3.2), or if the user chooses not to activate the reservation feature. In this case, the broker resorts to predicting the waiting time from the current load of the resource. This alternative prediction tends to be very coarse due to the complex nature of batch system scheduling algorithms and the limited information available to the broker about other queuing jobs.

3.2. Advance resource reservations

The advance reservation feature makes it possible to obtain a guaranteed start time for a job, giving several advantages. It makes it possible to meet deadlines for time-critical jobs and to coordinate the job with other activities. In this section, we present two alternative approaches for supporting advance reservations, one implemented as an extension to the ARC GridFTP server and the other a service-based reservation framework.

The reservation mechanism-based on GridFTP is implemented as an extension to the job management plug-in in the ARC GridFTP server. The reservation protocol supports two operations: requesting a reservation and releasing a reservation. The reservation request contains the start time and requested duration of the reservation and the required number of CPUs. Upon receiving a reservation request from the broker, the GridFTP server on the resource authorizes the requestor. After authorizing the user, the job management plug-in of the GridFTP server invokes a script to request a reservation from the local scheduler. If the scheduler accepts the request and creates the reservation, the GridFTP server returns a unique identifier and the start time of the reservation to the broker. If no reservation can be created, a message indicating failure is returned. The GridFTP server saves the reservation identifier and a copy of the user's proxy for every successful reservation, enabling subsequent authorization of the user who made the reservation. To release a reservation, the broker uploads a release message containing the reservation identifier and the GridFTP server confirms that the reservation is released.

The service-based framework for creating and managing reservations builds on OGSI-compliant Grid services [30,31] and is implemented using the Globus Toolkit version 3 [32]. The framework consists of two services: the *ReservationFactory*, for creating reservations and the *Reservation*, which is used for controlling and monitoring created reservations. These services implement the following subset of the functionality described in [33]; two-phase reservations, i.e. reservations that have soft-state and are released shortly after their creation unless they are confirmed, and a reservation architecture not locked to a specific resource but supporting reservations of multiple resource types (computers, networks, disks, etc.). The implementation provides the resource type independence, but reservation plug-in components are currently only supported for computers.

The ReservationFactory is independent of the local reservation system and uses a set of *reservation managers* to handle interactions with the reservation management system for a specific resource type (computer, network, etc.). The operation exposed by the ReservationFactory is createReservation, which takes a set of general and a set of resource specific parameters as its arguments. The general parameters include the resource type requested, a start time window specifying the range of acceptable start times for the reservation, the time when the reservation should be released unless it is confirmed, and a flag indicating whether the reservation is malleable. The start time for a malleable reservation may be altered by the local scheduler as long as it is kept within the specified start time window. Resource utilization typically decreases if advance reservations are used [8], but this impact can be reduced if the reservations are malleable [34].

The resource-specific parameters have no fixed type and can be used to describe any type of requirement, e.g. the number of CPUs to be reserved on a cluster. The createReservation operation returns the exact start time of the reservation and a local reservation identifier. When createReservation is invoked, the ReservationFactory forwards the incoming request to the reservation manager of the type specified in the request. For computer reservations, the actual interaction with the local reservation manager is performed similarly as in the GridFTP-based reservation system. Moreover, a copy of the user's proxy certificate is stored for later authorization.

One instance of the Reservation service is created for each reservation. The Reservation service exposes operations for querying the status of the reservation, confirming a reservation and cancelling a reservation (destroying the service). The last operation is reused from its implementation in [32].

Below, we outline a short summary of advantages and disadvantages of the two approaches to implement advance reservations.

The GridFTP-based reservation framework is lightweight and has good performance. It is also easy to deploy, as the ARC GridFTP server is already installed on the resources and is used by the broker as described in Section 2.1. On the other hand, this solution is non-standard and can hence only be used in the ARC middleware and only to reserve computational resources. Furthermore, there is no support for two-phase reservations which causes a waste of resources if the broker for some reason is unable explicitly to release a created reservation that will never be used.

The service-based reservation framework is a more general and flexible solution that is not limited to computational resources and deployment in ARC. It is also based on standard Web services technology instead of a custom plug-in to the ARC GridFTP server, and it supports two-phase reservations. The disadvantages with the service-based version include the overhead associated with service invocation and that the service-based framework requires installation of several additional software components, on both the Grid resources and in the client (broker).

3.2.1. Job submission with a reservation

If a reservation is successfully created on the selected resource, the broker adds the received reservation identifier to the xRSL job description before submitting the job request to the resource.

Before the Grid manager submits the job to the local scheduler, a reservation authorization plug-in script analyzes the job description and detects the reservation identifier. The script inspects the saved proxy files and their associated reservation identifiers to ensure that the specified reservation exists. Furthermore, the script compares the proxy used to submit the job with the one used to create the reservation. The job request is denied unless the specified reservation exists and is created by the job submitter.

After job completion, the Grid manager may remove the reservation, allowing the user to run only the requested job. Alternatively, resources may permit the user to submit more jobs within the same reservation once the first job is completed. The configuration of the Grid manager plug-in script and the local batch system determines the policy to be used.

The advance reservation feature requires that a reservation capability is provided by the local scheduler. The current

implementation supports the Maui scheduler [35], although any local scheduler may be used (see, e.g., [7]). Support for other schedulers than Maui can easily be added by adapting the scripts that create and release reservations from the local batch system.

3.3. Job queue adaptation

Network load and batch queue sizes may change rapidly in a Grid. New resources may appear and others become unavailable. The load predictions used by the broker as a basis for resource selection can quickly become outdated. Nevertheless, more recent information will always be available as Grid resources periodically advertise their state. To compensate for this, the broker has functionality to keep searching for better resources once the initial job submission is done. If a new resource that is likely to result in an earlier job completion time is found (taking into account all components of the TTD, including file restaging and job restart), the broker cancels the job and resubmits it to the new resource. This procedure is repeated until the job starts to execute on the currently selected resource. The job queue adaptation procedure can be viewed as the simplest form of Grid job migration, studied by, e.g., [22].

4. User interface extensions

We have extended the standard ARC user interface with some new options and added some new attributes to the ARC xRSL, in order to make the new features available to users.

4.1. Benchmarks

In order to make use of the feature of benchmark-based execution time prediction, the user must provide relevant benchmark information as described by the following example. Assume that the user knows that the performance of the application `my_app` is well characterized by the NAS benchmarks LU, BT and CG. For each of these benchmarks, the user can specify a benchmark result and an expected execution time on a system corresponding to that benchmark result. Notably, the expected execution time must be specified for each benchmark, as the benchmark results may be from different reference computers. This is specified using the new xRSL attribute `benchmarks`.

Fig. 3 illustrates how to specify that the application requires 65 min to execute on a cluster where the results for the NAS LU and BT benchmarks class C are 250 and 200, respectively. The estimated execution time is 50 min on an (apparently different) cluster where the CG benchmark result is 90.

In order to use benchmark-based execution time predictions, the information advertised by each cluster (about hardware, software, current state, etc.) must be extended with benchmark results for that cluster. Users cannot, however, know in advance what benchmarks are advertised by the clusters. To simplify the usage of benchmark-based execution time predictions, there is an additional client tool for discovering

```
&(executable = my_app)
(stdin = my_app.in)
(stdout = my_app.out)
(benchmarks =
    (nas-lu-c 250:65)
    (nas-bt-c 200:65)
    (nas-cg-c 90:50))
```

Fig. 3. Sample xRSL request including benchmark-based execution time predictions.

```
&(executable = my_program)
 (arguments = params input)
 (stdout = log)
 (inputfiles =
    (params gsiftp://host1/file1)
    (input http://host2/file2))
 (outputfiles =
    (results gsiftp://host3/my_program.results)
    (all_data gsiftp://host3/my_program.data)
    (log gsiftp://host4/my_program.log))
 (outputfilesizes =
    (results 230MB)
    (all_data 5GB))
```

Fig. 4. Sample xRSL request with information required to estimate file transfer times.

all benchmarks advertised by any of the clusters. This client performs resource discovery just as the broker, but instead of submitting a job, the client outputs a list of available benchmarks. For each benchmark, a list of clusters advertising the benchmark is printed with the benchmark result for each machine. From this list, a user can find a reference benchmark result for a machine where the user's job has been executed previously.

4.2. Network transfers

In the example in Fig. 4, the job involves the transfer of large input and output files. The broker determines the actual sizes of the input files when estimating the transfer time for these. The new, optional, xRSL attribute `outputfilesizes` enables the user to provide an estimate of the size of the job output. A typical user runs the same application many times and will normally, with time, be able to provide very accurate estimations of job output size. As shown in Fig. 4, the user does not have to include size estimates for all output files in the `outputfilesizes` relation. File size estimates can be specified in bytes or with any of the suffixes kB, MB or GB.

4.3. Command line options

In addition to the xRSL extensions, the broker supports some new command line options. The option `-A` is used to request the broker to perform queue adaptation. The reservation feature is

activated using the option `-R`. The option `-S` is used to build a pipeline between jobs, so that output from one job is used as input to the next.

5. Future work

Current and future directions of this research include the development of a service-oriented stand-alone resource broker and job submission service with the same basic functionality as the current tool [13,14]. This includes an investigation of how various (emerging) Grid and Web services standards can be used to improve the portability and interoperability of the job submission service, e.g., in order to facilitate cross-middleware job submission. We also plan to complement the service with additional general components, e.g., for job monitoring and control. Additional topics currently being addressed include the design and analysis of efficient algorithms for resource co-allocation.

6. Concluding remarks

The presented resource broker is developed with a focus on the ARC middleware and the NorduGrid and SweGrid production environments. Some of its brokering algorithms are currently in production use in both environments. The broker includes support for making advance resource reservations and it selects resources based on benchmark-based execution time estimates and network performance predictions. The broker is a built-in component of the user's job submission software, and is hence a decentralized user-oriented broker acting with no need for global control, entirely basing its decisions on the dynamic information provided by the resources.

Acknowledgments

We thank Peter Gardfjäll and Åke Sandgren for fruitful discussions. We are also grateful to the anonymous referees, whose constructive comments have improved the presentation of this work.

References

[1] J. Brooke, D. Fellows, Draft discussion document for GPA-WG — abstraction of functions for resource brokers. http://www.ogf.org/Meetings/ggf7/drafts/GGF7_rbdraft.pdf, May 2006.

[2] J. Schopf, Ten actions when Grid scheduling, in: J. Nabrzyski, J. Schopf, J. Węglarz (Eds.), Grid Resource Management: State of the Art and Future Trends, Kluwer Academic Publishers, 2004, pp. 15–23 (Chapter 2).

[3] R. Yahyapour, Considerations for resource brokerage and scheduling in Grids, in: G.R. Joubert, W.E. Nagel, F.J. Peters, W.V. Walter (Eds.), Parallel Computing: Software Technology, Algorithms, Architectures and Applications, PARCO 2003, Dresden, Germany, 2004, pp. 627–634.

[4] I. Foster, J. Schopf, L. Yang, Conservative scheduling: Using predicted variance to improve scheduling decisions in dynamic environments, in: Proceedings of the ACM/IEEE SC2003: International Conference for High Performance Computing and Communications, 2003, pp. 31–46.

[5] K. Ranganathan, I. Foster, Simulation studies of computation and data scheduling algorithms for data Grids, Journal of Grid Computing 1 (1) (2003) 53–62.

[6] W. Smith, I. Foster, V. Taylor, Predicting application run times using historical information, in: D. Feitelson, L. Rudolph (Eds.), Job Scheduling Strategies for Parallel Processing, in: LNCS, vol. 1459, Springer-Verlag, Berlin, 1999, pp. 122–142.

[7] J. MacLaren, Advance reservations state of the art. http://www.fz-juelich.de/zam/RD/coop/ggf/graap/sched-graap-2.0.html, May 2006.

[8] W. Smith, I. Foster, V. Taylor, Scheduling with advance reservations, in: 14th International Parallel and Distributed Processing Symposium, IEEE, Washington, Brussels, Tokyo, 2000, pp. 127–132.

[9] K. Czajkowski, I. Foster, C. Kesselman, V. Sander, S. Tuecke, SNAP: A protocol for negotiating service level agreements and coordinating resource management in distributed systems, in: D. Feitelson, L. Rudolph, U. Schwiegelshohn (Eds.), Job Scheduling Strategies for Parallel Processing, in: LNCS, vol. 2537, Springer-Verlag, Berlin, 2002, pp. 153–183.

[10] I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, A. Roy, A distributed resource management architecture that supports advance reservations and co-allocation, in: 7th International Workshop on Quality of Service, IEEE, Washington, Brussels, Tokyo, 1999, pp. 27–36.

[11] NorduGrid. http://www.nordugrid.org, May 2006.

[12] SweGrid. http://www.swegrid.se, May 2006.

[13] E. Elmroth, J. Tordsson, An interoperable, standards-based Grid resource broker and job submission service, in: H. Stockinger, R. Buyya, R. Perrott (Eds.), First International Conference on e-Science and Grid Computing, IEEE CS Press, 2005, pp. 212–220.

[14] E. Elmroth, J. Tordsson, A standards-based Grid resource brokering service supporting advance reservations, coallocation and cross-Grid interoperability, Department of Computing Science and HPC2N, Umeå University, SE-901 87 Umeå, Sweden, December 2006 (submitted for journal publication).

[15] M. Ellert, M. Grønager, A. Konstantinov, B. Kónya, J. Lindemann, I. Livenson, J.L. Nielsen, M. Niinimäki, O. Smirnova, A. Wäänänen, Advanced Resource Connector middleware for lightweight computational Grids, Future Generation Computer Systems 23 (2) (2007) 219–240.

[16] OpenLDAP. http://www.openldap.org, May 2006.

[17] OpenSSL. http://www.openssl.org, May 2006.

[18] I. Foster, C. Kesselman, Globus: A metacomputing infrastructure toolkit, International Journal of Supercomputer Applications 11 (2) (1997) 115–128.

[19] Globus. http://www.globus.org, May 2006.

[20] P. Gardfjäll, E. Elmroth, L. Johnsson, O. Mulmo, T. Sandholm, Scalable Grid-wide capacity allocation with the SweGrid Accounting System (SGAS), Department of Computing Science and HPC2N, Umeå University, SE-901 87 Umeå, Sweden, October 2006 (submitted for journal publication).

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

[22] M. Litzkow, M. Livny, M. Mutka, Condor — a hunter of idle workstations, in: Proceedings of the 8th International Conference of Distributed Computing Systems, 1988, pp. 104–111.

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

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

[25] C. Anglano, T. Ferrari, F. Giacomini, F. Prelz, M. Sgaravatto, WP01 report on current technology. http://server11.infn.it/workload-grid/docs/DataGrid-01-TED-0102-1_0.pdf, May 2006.

[26] K. Krauter, R. Buyya, M. Maheswaran, A taxonomy and survey of Grid resource management systems for distributed computing, Software - Practice and Experience 15 (32) (2002) 135–164.

[27] L. Perini, (Coordinator), Atlas Grid computing. http://atlas.web.cern.ch/Atlas/GROUPS/SOFTWARE/OO/grid/, November 2006.

[28] A. Lastovetsky, Parallel Computing on Heterogeneous Networks, 1st ed., Wiley-Interscience, 2003.
[29] R. Wolski, Dynamically forecasting network performance using the Network Weather Service, Journal of Cluster Computing 1 (1) (1998) 119–132.
[30] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maguire, T. Sandholm, P. Vanderbilt, D. Snelling, Open Grid Services Infrastructure (OGSI) version 1.0. http://www.globus.org/alliance/publications/papers/Final_OGSI_Specification_V1.0.pdf, May 2006.
[31] I. Foster, C. Kesselman, J. Nick, S. Tuecke, Grid services for distributed system integration, Computer 35 (6) (2002) 37–46.
[32] J. Gawor, T. Sandholm, Globus toolkit 3 core — a Grid service container framework. http://www-unix.globus.org/toolkit/3.0/ogsa/docs/gt3_core.pdf, May 2006.
[33] A. Roy, V. Sander, Advance reservation API. http://www.ogf.org/documents/GFD.5.pdf, May 2006.
[34] U. Farooq, S. Majumdar, E.W. Parsons, Impact of laxity on scheduling with advance reservations in Grids, in: MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, IEEE Computer Society, Washington, DC, USA, 2005, pp. 319–324.
[35] Supercluster.org. Center for HPC Cluster Resource Management and Scheduling. http://www.supercluster.org, May 2006.

Erik Elmroth received his Ph.D. in 1995 and is currently Associate Professor in Computing Science, Umeå University. He is deputy head of the Department of Computing Science and deputy director for the High Performance Computing Center North (HPC2N). At the Swedish Research Council (VR), he is a member of the Committee for Research Infrastructures (KFI) and vice-chair of its expert panel on e-Science infrastructures. International experiences include a year as post-doctoral researcher at NERSC, Lawrence Berkeley National Laboratory, University of California, Berkeley, and one semester as Visiting Scientist at Massachusetts Institute of Technology (MIT), Cambridge, MA. His broad interests in algorithms and software for scientific computing include Grid Computing, parallel computing, algorithms for managing memory hierarchies, linear algebra library software, and ill-posed eigenvalue problems. He is co-winner of the SIAM Linear Algebra Prize 2000, for the most outstanding linear algebra publication worldwide during the preceding three-year period.

Johan Tordsson is a Ph.D. student at the Department of Computing Science, Umeå University. He received his Master and Licentiate degrees from Umeå University in 2004 and 2006, respectively. His research interest is Grid computing, including standardization efforts, resource brokering, performance estimation, quality of service and automated negotiation.

II

Paper II

An Interoperable, Standards-based Grid Resource Broker and Job Submission Service*

E. Elmroth and J. Tordsson

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

Abstract: We present the architecture and implementation of a Grid resource broker and job submission service, designed to be as independent as possible of the Grid middleware used on the resources. The overall architecture comprises seven general components and a few conversion and integration points where all middleware-specific issues are handled. The implementation is based on state-of-the-art Grid and Web services technology as well as existing and emerging standards (WSRF, JSDL, GLUE, WS-Agreement), Features provided by the service include advance reservations and a resource selection process based on a priori estimations of the total time to delivery for the application, including a benchmark-based prediction of the execution time. The general service implementation is based on the Globus Toolkit 4. For test and evaluation, plugins and format converters are provided for use with the NorduGrid ARC middleware.

* By permission of IEEE Computer Society Press.

An Interoperable, Standards-based Grid Resource Broker and Job Submission Service

Erik Elmroth and Johan Tordsson
Dept. of Computing Science and HPC2N
Umeå University, SE-901 87 Umeå, Sweden
{elmroth, tordsson}@cs.umu.se

Abstract

We present the architecture and implementation of a Grid resource broker and job submission service, designed to be as independent as possible of the Grid middleware used on the resources. The overall architecture comprises seven general components and a few conversion and integration points where all middleware-specific issues are handled. The implementation is based on state-of-the-art Grid and Web services technology as well as existing and emerging standards (WSRF, JSDL, GLUE, WS-Agreement). Features provided by the service include advance reservations and a resource selection process based on a priori estimations of the total time to delivery for the application, including a benchmark-based prediction of the execution time. The general service implementation is based on the Globus Toolkit 4. For test and evaluation, plugins and format converters are provided for use with the NorduGrid ARC middleware.

1. Introduction

The resource broker and job submission components are vital for any Grid computing infrastructure, as their functionality and performance to a large extent determine the user's experience of the Grid. In all, these components have to identify, characterize, evaluate, select, and allocate the resources best suited for a particular application. The brokering problem is complicated by the heterogeneous and distributed nature of the Grid as well as the differing characteristics of different applications. To further complicate matters, the broker typically lacks total control and even complete knowledge of the state of the resources.

Typically, resource brokers are closely integrated with, or at least heavily dependent on, some particular Grid middleware, with popular solutions ranging from brokering components being part of the job submission client to centralized Grid-schedulers not that different from traditional batch system schedulers [16, 7, 20, 15, 5, 13]. Hence, it is normally non-trivial to migrate a broker from one middleware to another, or to adjust it to simultaneously work with resources running different middlewares.

This contribution presents an architecture and an implementation of a general service for Grid resource brokering and job submission. The service is general in the sense that it can be used with different Grid middlewares, with middleware-specific issues concentrated to minor components. These components are used for format conversions in interactions with clients and information systems as well as for middleware-specific interaction with resources.

The proposed broker and job submission service rely heavily on Grid and Web services standards, including JSDL, WSRF, WS-Agreement, and GLUE (see Section 2), and are implemented using Globus Toolkit 4 (GT4) [10]. Middleware-specific interfaces are provided for the NorduGrid ARC software [8, 14], which is based on the Globus Toolkit 2 (GT2). Tests and evaluation have been performed on SweGrid [18] and NorduGrid [14] resources.

The brokering scenario addressed by our solution is a decentralized broker that acts on behalf of the user in order to allocate the resources that best fulfill the user's request. Hence, the broker does not take any globally controlling role and works independently of any other broker or job submission software interfacing the same resources. All decisions made by the broker are based on the user's requests and the information (including negotiation) it extracts from the resources and information services. Notably, the broker may be used simultaneously by multiple users, but the brokering scenario remains as described above.

The broker aims at identifying the set of resources that minimizes the *Total Time to Delivery (TTD)*, or part thereof, for each individual job submission [9]. In order to do this, the broker makes an a priori estimation of the whole or parts of the TTD for all resources of interest before making the selection. The TTD estimation includes performing a *benchmark-based execution time estimation*, *estimating file*

Proceedings of the First International Conference on e-Science and Grid Computing (e-Science'05)
0-7695-2448-6/05 $20.00 © 2005 IEEE

transfer times, and performing *advance reservations* of resources in order to obtain a guaranteed batch-queue waiting time. For resources not providing all information required or a reservation capability, less accurate estimations are performed.

The rest of this paper is organized as follows. Section 2 introduces some standards and technologies used, Section 3 gives an in-depth description of the system and Section 4 describes the resource selection algorithms used. Section 5 illustrates the integration of the system with an existing Grid middleware, whereas sections 6 and 7 contain a performance evaluation and conclusions, respectively.

2. Background and standards used

The presented brokering and job submission architecture makes extensive use of existing and proposed Grid and Web service standards not only for interaction with other components but also internally. The most important standards used are briefly presented below.

2.1. JSDL

The Job Submission Description Language (JSDL) proposed by the Global Grid Forum (GGF) describes the configuration of computational jobs and their requirements on the resources that executes them. It is the result of the JSDL working group's attempts to create a standardized job description language, simplifying interoperability between existing resource management systems [3].

In our contribution, the JSDL is used to express job requests sent to the job submission module by clients.

2.2. WSRF

The Web Services Resource Framework (WSRF) [11], defines a relationship between stateful resources and Web services. This relationship is modelled using a construct called a WS-Resource. An endpoint reference addresses a Web service, and may also identify one of the WS-Resources associated with that service.

The WSRF consists of five specifications, including the following. WS-ResourceProperties defines the type and value of the WS-Resource's state as viewable through a Web service interface. The WS-ResourceLifetime specification defines lifecycle management of WS-Resources, including creation and destruction (immediate or scheduled for later). WS-BaseFault defines a base type for fault handling in Web services, which increases consistency.

In our work, WSRF, and more specifically, WS-Resources are used to represent jobs and reservations (agreements). Information about submitted jobs and created reservations is modelled using WS-ResourceProperties. The lifetime management mechanisms defined in WS-ResourceLifetime are used to implement soft-state, two-phase reservations. WS-BaseFault is used for error messages.

2.3. WS-Agreement

WS-Agreement is a GGF standard proposal, which makes it possible for an *agreement initiator* and an *agreement provider* to enter an agreement. This agreement specifies service level objectives associated with the use of one or more Web services. The WS-Agreement standard does not specify any domain-specific terms describing the service level objectives, but is rather intended for use with any type of Web service. Service domain-specific terms are expected to be added in extensions for each service domain of interest [2].

The basic operation of WS-Agreement is straightforward. Initially, the agreement initiator retrieves an *agreement template* (prefilled contract) from the agreement provider. The initiator fills out the relevant parameters in the template and sends the resulting *agreement offer* in a request to the agreement provider. Upon granting the offer, the agreement provider creates a WS-Resource representing the agreement, and returns an endpoint reference to this WS-Resource to the agreement initiator.

The AgreementFactory porttype stores the agreement templates and exposes an operation for requesting an agreement. The Agreement porttype exposes no operation, it only holds the WS-Resources modelling created agreements. A third porttype, the AgreementState, is used to monitor the fulfillment of the agreement. Notably, WS-Agreement neither defines a protocol for agreement negotiation, nor states how agreements should be signed.

In our work, WS-Agreement is used to negotiate and represent advance reservations for batch systems.

2.4. GLUE

The Grid Laboratory Uniform Environment (GLUE) project [1] defines an information model for describing Grid resources, targeting core services such as resource discovery and monitoring. Resource discovery services benefit from an extensive list of resource characteristics. For monitoring, state information describing load and availability is defined. The GLUE model (version 1.2) describes computing elements, storage elements, and mappings relating these. The GLUE project targets a model usable by different technologies. Current implementations include LDAP schemas for GT2 and XML schemas for GT4.

In our job submission service, the GLUE format is used to represent resource information gathered during resource discovery.

3. Architecture

The proposed brokering and job submission framework is based on a general architecture with seven components and an implementation of the WS-Agreement specification. The framework is complemented with a job submission client and some middleware-specific components, currently available for the NorduGrid ARC middleware.

The job submission service itself is implemented using GT4 [19, 10], and does, just like GT4, make extensive use of Axis [4]. Presently, plugins for reservations are implemented for the Maui scheduler [12].

Below we give an architecture overview, followed by more detailed descriptions of each of the modules, including some discussions on design considerations.

3.1. Overview

The job submission module consists of seven components: the *InformationFinder* performs resource discovery and retrieves resource information; the *Broker* performs resource selection; the *Reserver* negotiates advance reservations; the *DataManager* handles file transfers; the *Dispatcher* sends job requests to the resource; the *Submitter* coordinates the work of the five first modules and finally the *JobSubmissionService* which stores information about submitted jobs and provides a Web service interface to the job submission module. In addition to these components, there is a user client for sending job requests to the JobSubmissionService. The system also includes an implementation of the WS-Agreement specification, hosted on the Grid resource.

Figure 1 gives an overview of the modules. Their interactions and main operations are the following. Upon receiving a job request from the client, the JobSubmissionService passes the job description along with any optional parameters to the Submitter. The Submitter first invokes the Broker to validate the job description. Then, the InformationFinder is used to retrieve a list of available Grid resources. After receiving this list, the Submitter calls the Broker to filter out unsuitable resources and to rank the suitable ones. The ranking procedure may include creating advance reservations, which is handled by the Reserver. If required, the DataManager is then invoked to stage input files. Then, the Submitter uses the Dispatcher to submit the job to the selected resource and returns the obtained job identifier to the JobSubmissionService.

The JobSubmissionService creates a stateful resource (WS-Resource) storing information associated with the job. In the final step, the job identifier is returned to the client.

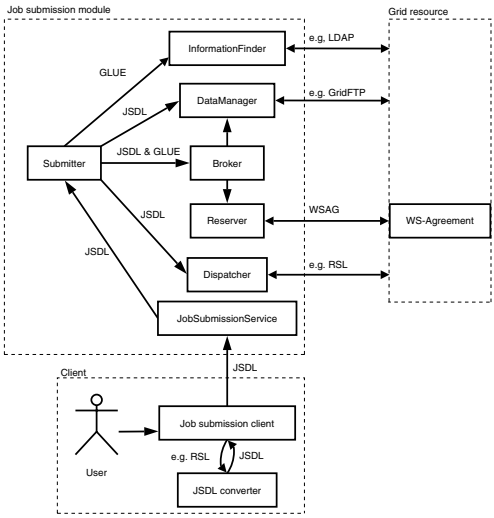


Figure 1. Architecture overview showing components, hosts, and information flow. The boxes show the modules and the dashed lines denote the different hosts.

3.2. Modules

This section presents the finer details of the modules in the system.

JobSubmissionService. The JobSubmissionService is the only component in the job submission module accessible by clients. It exposes one operation, *SubmitJob*, through its Web service interface. The parameters for this operation are the JSDL job description, and optionally, a document describing the user's job preferences and/or a list of URLs of index servers, e.g., GT2 GIISes or GT4 Index Services, to contact during resource discovery.

Notably, the JobSubmissionService does not expose any other operation than job submission. Further job management, such as job monitoring and control are beyond the scope of this service. The JobSubmissionService stores stateful information about each submitted job, including the job identifier, the job description and, if applicable, information about an advance reservation created for the job.

A WS-Resource is created for each successfully submitted job, with job information stored as resource properties. By querying the resource properties, other job management tools can retrieve the job identifier in order to monitor and control the execution of the job.

As the JobSubmissionService typically is invoked by a

user, a job submission client is provided. The arguments to the client include those passed in a job request to the JobSubmissionService, i.e., the mandatory job description and the optional job preferences document and index server URLs. In addition to these parameters, a user can specify which JobSubmissionService to contact. The client supports a plugin structure enabling translations to JSDL from any native job description language preferred by the user.

InformationFinder. The purpose of the InformationFinder is to discover what Grid resources are available and to retrieve more detailed information about each resource. The input to this module is a list of index server URLs.

The InformationFinder first performs resource discovery by querying each index server, which due to their hierarchical organization may require some recursive invocations. Then, each identified resource is queried in more detail. Both static and dynamic information about the resource is retrieved, including hardware and software configuration and current load. The InformationFinder also retrieves usage policies, allowing it to discard resources where the user is not authorized to submit jobs.

Unless the information retrieved in the detailed queries follows the GLUE format, it is converted by the InformationFinder before being returned to the Submitter. To improve performance, the resource discovery, the information retrieval, and the conversion to GLUE format are each performed in parallel. A fixed size thread pool is used to control the degree of parallelism, thus avoiding overloading the hosting environment. In order to reduce the resource discovery overhead in situations where the same resources are repeatedly queried during a short period of time, the retrieved resource information is stored in a time-limited cache.

Broker. The Broker strives to select the best resource for each incoming job request. What makes a resource the "best resource" depends on the characteristics of the job and the resources, as well as on the user's preferences.

The Broker provides three operations. Validation of job description ensures that the description contains all required attributes, e.g., the application to run. Resource filtering guarantees that only resources fulfilling the job's requirements on architecture, disk, memory etc. are considered for submission. The most complex operation, resource ranking, ranks the resources according to their suitability for executing the job and reorders the resource list accordingly. Two interfaces are defined to facilitate this operation.

The *predictor* interface is used for the estimation of how long time a certain task associated with the job would require if performed by a certain resource. The four tasks considered for time estimation are: staging input files to the resource; waiting for resource access, e.g., in a batch queue; executing the job on the resource; and staging output files. These four tasks make up the Total Time to Delivery (TTD) for the Grid job. The algorithms used for TTD estimations are presented in [9] and reviewed in Section 4.

The *selector* interface is used for the actual ranking of the resources. The ranking is done using some, possibly all, of the predictors. Currently, the Broker contains two selectors. The earliest start selector ranks resources based on the stage in and wait predictors, in order to achieve an as early job start as possible. In contrast, the earliest completion selector tries to minimize the TTD, and hence achieve the earliest possible job completion.

Resource ranking may include the time-consuming task of negotiating advance reservations. To improve performance, resources are ranked in parallel, using a thread pool similar to the one in the InformationFinder.

DataManager. The DataManager is responsible for all data management tasks that relate to job submission. The module defines an interface for staging of files to and from Grid resources. This interface also defines the resolution of physical location(s) of replicated files, which is useful if the Grid middleware supports file replication. Another operation defines the prediction of the duration of file transfers, which can be implemented if the underlying infrastructure supports either network reservations or bandwidth performance predictions. The DataManager also implements an operation for determining the sizes of physical files. This operation is used as a last resort for predicting the duration of file transfers when none of the more sophisticated mechanisms mentioned above are available.

Reserver. The Reserver includes a client API for reserving CPUs at computational resources in advance. These CPUs are reserved for a certain duration and with either a fixed start time or an interval of allowed start times.

The three operations defined in the Reserver API are creation of a temporary reservation, confirmation of a temporary reservation, and cancellation of a reservation (temporary or confirmed). Temporary reservations are released shortly after their creation unless they are confirmed. These operations are implemented using calls to the WS-Agreement module.

The Reserver also contains a repository of created reservations. After a job is successfully submitted to a resource, the Submitter examines the repository and confirms the reservation on the selected resource and cancels any other reservation created for the job.

WS-Agreement. This module includes implementations of the AgreementFactory and Agreement porttypes defined by the WS-Agreement specification. Unlike the other modules, which are hosted on the machine running the JobSubmissionService, WS-Agreement is deployed on the Grid resource.

Note that our current implementation does not include the AgreementState porttype. For our target domain, advance reservations of CPUs, monitoring the state makes little sense. A created reservation will, short of resource failure, fulfill the guarantees specified in the agreement.

Two interfaces are defined in order to guarantee that the WS-Agreement implementation is agnostic of the service domain for which the agreements are created. Both interfaces must be implemented when using WS-Agreement for a specific service, and any service-specific operation has to be placed within the implementations of these interfaces.

The AgreementDecisionMaker interface determines whether to grant or deny an agreement offer, returning an AgreementDecision, which, in addition to the actual decision, contains any domain specific context associated with the created agreement. The AgreementDecisionMaker concept first appeared in Cremona [6].

The AgreementResourceHelper constructs domain specific agreement terms for inclusion in the resource property document of the WS-Resource representing the agreement.

To the best of our knowledge, there exists no earlier work using WS-Agreement to model batch queue reservations. For this reason, a language for describing reservations against computational resources is defined. Each agreement offer (reservation request) contains the duration of the reservation and the requested number of CPUs. Furthermore, a start time window specifying earliest and latest allowed job start is included. A flag named flexible specifies whether the start time of the reservation may be moved within the start time window by the local batch system. If allowing this, backfilling can be performed more efficiently, resulting in increased utilization [17].

The AgreementFactoryService passes an incoming agreement offer to the ReservationDecisionMaker, which executes a plugin creating the earliest possible reservation within the start time window specified in the offer. If no reservation can be created within this window, the offer is denied. After the ReservationDecisionMaker has granted the offer, the AgreementFactoryService creates a WS-Resource and invokes the ReservationResourceHelper to create resource properties for this WS-Resource. The resource properties include an identifier for the reservation, the exact reservation start time and the parameters in the agreement offer. Finally, the endpoint reference to the WS-Resource is returned to the agreement initiator (the Reserver). Figure 2 shows the interactions between the Reserver and the WS-Agreement services.

Dispatcher. The Dispatcher is responsible for sending the job request to the selected resource. While this task may seem trivial, the job description may first have to be translated (back) from JSDL to the job description language understood by the resource. Furthermore, the mechanism used for sending the job request to the resource depends on the

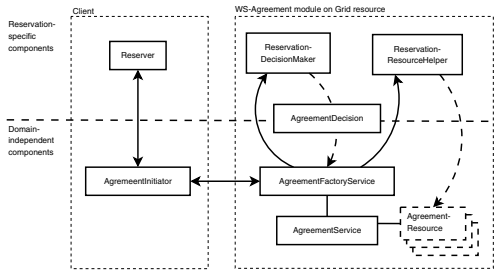


Figure 2. Interactions between general WS-Agreement components and the reservation specific modules.

Grid middleware used on the resource. Possible approaches include invocation of a Web service, as used by Globus WS-GRAM, and interaction with a GridFTP server, which is the mechanism used by NorduGrid ARC. Due to these significant differences, the Dispatcher contains no code common for all middlewares, but rather defines a general interface for dispatching jobs. This interface includes the job dispatch operation, which takes three arguments, the GLUE information about the selected resource, the job description (in JSDL format), and optionally, information about an advance reservation created for the job. The interface also defines the translation of job descriptions from JSDL to the native job description language of the used Grid middleware.

Submitter. The Submitter coordinates the job submission process. When a job request is passed from the JobSubmissionService, the Submitter invokes the Broker to validate the job description. If the description is valid, the Submitter calls the InformationFinder to retrieve an updated list of resource information. Once the resource list is updated, the Broker is invoked twice by the Submitter. First for filtering out inadequate resources, then for reordering the remaining ones after their suitability for executing the job. This second step may include requesting advance reservations, a task handled by the Reserver. Unless the Grid middleware performs file staging, the Submitter next invokes the DataManager to stage input files. It then calls the Dispatcher to send the job request to the most suitable resource. If one of these two operations fails, the Submitter retries with the second most suitable resource etc., until the job either is successfully submitted or all submission attempts fail, the latter causing an error message to be returned to the client. After completing these tasks, the Submitter returns a middleware-specific job identifier to the JobSubmissionService.

It may seem superfluous to separate the Submitter and the JobSubmissionService into two layers. We do however believe that the separation of the job submission process

handled by the Submitter and the management of stateful resources performed by the JobSubmissionService is beneficial. This allows the construction of alternative Submitters, e.g., for coallocation or workflow scheduling.

4. Resource Brokering Algorithms

The resource broker selects the resources that gives the minimum predicted TTD (or part thereof) for the application. The algorithms for predicting the TTD depend on the support provided by the resources and the optional information supplied by the user. Below, we review the algorithms used for predicting the TTD, originally presented in [9], and describe what optional information a user can provide in order to improve the brokering process.

4.1. A priori estimation of TTD

The TTD for a Grid job includes the times for (1) staging in the executable and the input files to the resource, (2) waiting in the batch queue, (3) executing the application, and (4) staging output files to their requested location(s). The Broker presented in Section 3 makes use of one predictor for each of these tasks. The two provided selectors make use of the first two and all four predictors, respectively.

If the DataManager provides support for predicting file transfer times or for resolving physical file locations and determining file sizes (see Section 3), these features are used by the predictors of (1) and (4) for estimating file staging times. If no such support is available, e.g., depending on the information provided by the Grid middleware used, these predictors make use of the file transfer times optionally provided by the user. Notably, the time estimate for stage in is important not only for predicting the TTD but also for coordinating the start of the execution with the arrival of the executable and the input files if an advance reservation is performed.

The most accurate prediction of the batch queue waiting time (2) is obtained by using advance reservations, which gives a guaranteed job start time. If the resource does not support advance reservations or the user chooses to deactivate this feature, less accurate estimates are made from the information provided by the resource about current load. This estimation, however, does not take into account the actual scheduling algorithm used by the batch system.

The prediction of (3) is performed through a benchmark-based estimation of execution time that takes into account both the performance of the resources and the characteristics of the application. This estimation requires that the user provides the following information for one or more benchmarks with performance characteristics similar to that of the application: the name of the benchmark, the benchmark performance for some system, and the application's (predicted) execution time on that system. Using this information, the application's execution time is estimated on other resources assuming that the performance of the application is proportional to that of the benchmarks. For more information, e.g., how information about multiple benchmarks is used, see [9].

4.2. Optional input for brokering

In addition to the JSDL document describing the job, a user may include a job preferences document in the job request sent to the JobSubmissionService. This document contains both job requirements and additional information that may improve the resource selection process.

The job requirements are the preferred job objective, which can be either earliest job completion or earliest job start. The job start offset enables users to request that the job starts after a certain time, which can be expressed either as an absolute time or a relative offset from now. This feature can be used e.g., for debugging, demonstrations and coallocation purposes. Users can also specify a latest allowed job start, ensuring that the job either starts in time or is not submitted at all. Users with no strong requirements on the start or completion times of their jobs can specify that no reservation should be created for the job. Such jobs receive best-effort job start times, which facilitate improved resource utilization [17].

The job preferences document is also where a user may provide the additional information mentioned above, that allows the broker to improve the resource selection. This includes the predicted times for file staging and information used for benchmark-based execution time estimation.

Just as the job preferences document itself is an optional parameter in the job request message, all parts of the job preferences document are optional. However, by making use of this feature, expert users can benefit from their knowledge of the job characteristics.

An example of a user preference document is shown in Figure 3. In this document, a user specifies two benchmarks, nas-lu-c and specFP2000, as characteristic for the performance of the job. The user specifies the (possibly predicted) application execution times 60 and 45 minutes on machines where the results of these benchmarks are 450 and 750, respectively. The file stage out time is estimated to 10 minutes. Notably, file staging predictions are normally very inexact as the resource which to stage files to and from actually is unknown. Still, a rough approximation may often provide additional value in situations where the broker has no other information about network performance.

In the job requirement part, the user specifies the earliest allowed job start, and also states that the job may start no more than 30 minutes later than the earliest allowed start

time. The job objective is an as early job completion as possible.

```
<JobPreferences>
    <SchedulingHints>
        <Benchmark name="nas-lu-c"
            result="450" time="60"/>
        <Benchmark name="specFP2000"
            result="1750" time="45"/>
        <FileTransfers>
            <StageOutTime>10</StageOutTime>
        </FileTransfers>
    </SchedulingHints>
    <JobRequirement>
        <EarliestAbsoluteStart>
            2005-08-11T10:00:00Z
        </EarliestAbsoluteStart>
        <LatestRelativeStart>
            30
        </LatestRelativeStart>
        <JobObjective>
            EarliestJobCompletion
        </JobObjective>
    </JobRequirement>
</JobPreferences>
```

Figure 3. Example of a user job preferences document.

5 Configuration and Middleware Integration

The integration of the general job submission module with a particular Grid middleware requires some configuration and some customized components. Below we summarize what needs to be done, including the set up of the Submitter to interact with a specific middleware and how to configure the WS-Agreement module to support reservations of computational resources. Finally, we illustrate this by the steps taken in our integration with the NorduGrid ARC [14, 8].

The basic operation of the JobSubmissionService is controlled by the configuration of the Submitter. This configuration decides which plugin modules are used by the InformationFinder for discovering resources, for querying resources, and if needed, for converting the information retrieved to the GLUE format. The configuration file also controls which Dispatcher to use. These settings do, i.e., determine which Grid middleware to use when communicating with the Grid resources. As resource discovery can be rather time-consuming, it is possible to specify a timeout to use when discovering and querying resources. Further tuning of the performance of the InformationFinder can be done by adjusting the maximum number of threads to use in the thread pool. The configuration file also includes URLs to one or more default index servers, which are used in the resource discovery phase unless the user includes URL(s) to index server(s) in the job request. A configuration file containing the above described parameters is passed to the JobSubmissionService upon service startup.

The configuration of the WS-Agreement services determines which agreement template(s) to store in the AgreementFactoryService. Also included in the configuration is the DecisionMaker to use when determining whether to grant an agreement or not. The configuration file may also include a list of DecisionMaker initialization parameters. In the reservation scenario, these parameters are used to specify plugin scripts that invoke the local scheduler when creating and cancelling reservations. Support for other schedulers than the currently supported Maui scheduler can easily be implemented by creating new reservation plugin scripts and reconfiguring the AgreementFactoryService. The configuration file also specifies which ResourceHelper to use.

5.1. Integration with NorduGrid ARC

Here, we illustrate the integration of the brokering and job submission service with NorduGrid ARC, a middleware based on GT2 with some GT2-components replaced or modified.

A major difference between ARC and GT2 is the job management. In ARC, the server-side GT2 GRAM components (gatekeeper and job manager) are replaced by custom components, a GridFTP server and a Grid Manager, respectively [8]. Another difference is that even though ARC uses the GT2 MDS, this is done with customized schemas. Information advertised by a ARC GRIS describes users, jobs and resources (clusters and their job queues).

For integrating the job submission module with ARC, plugins are required for the Dispatcher and the InformationFinder. Furthermore, a few server-side scripts are required for managing the advance reservations created by the WS-Agreement components.

The hierarchal MDS structure used in ARC makes the resource discovery plugin in the InformationFinder straightforward. Starting from the list of GIISes provided on input, all GIISes are recursively queried for resources, each by a separate thread, without calling any GIIS more than once. Then, the resource query plugin requests information about the cluster and its queues, using LDAP. The result of these queries is objects providing an ARC-specific description of the resources, which are converted to the GLUE format by the converter plugin.

An ARC job submission client procedure includes uploading any locally stored input files to the resource (the Grid Manager handles stage in of non-local input files) before sending the job description to the GridFTP server of the resource. The ARC dispatcher plugin converts the JSDL job description to the GT2-style RSL used by ARC, and then modifies the job description to ensure that also local input files are staged by the Grid Manager. If an advance reservation is created for the job, the identifier of the reservation is

added to the job description before it is uploaded.

The implementation of WS-Agreement and the local scripts used by the ReservationDecisionMaker operates independently of ARC. However, a mechanism is required to associate the user creating the reservation with the one submitting the job. This is done by ensuring that the job submission and the creation of the reservation both are performed using proxies that originate from the same certificate.

A general infrastructure such as the job submission module can normally not capture all the features available in every single middleware, e.g., ARC. However, for use with ARC, the only noticeable shortcoming is that in ARC, files accessible through the Globus Replica Location Service (RLS) can serve as job input and output, whereas the job submission module currently only handles locally stored files and files stored at (Grid)FTP servers. Support for RLS may however be added in future versions of the job submission module.

6. Performance evaluation

The job submission module has been evaluated with respect to the service response time, i.e., the time required to submit a job, and the service throughput, i.e., number of jobs submitted to resources per minute.

The evaluation has been performed with the client and job submission module each running on a system equipped with one 2.8 GHz Intel P4 processor with 1 GB memory, Debian Linux Sarge and Globus Toolkit version 4.0.0. The WS-Agreement services were deployed on a system with a 667 MHz Intel P3 processor, 384 MB memory, Debian Linux Sarge, Globus Toolkit 4.0.0, Maui 3.2.6 and Torque 1.1.0. The Grid infrastructure used is a subset of NorduGrid and SweGrid, with in total 12 resources ranging from four to 388 CPUs (including the six 100 CPU clusters in SweGrid). These resources used the NorduGrid ARC middleware and were indexed by four GIISes, with one serving as higher level GIIS for the others. The LDAP information gathered from the resources was valid for 30 seconds, which hence became the cache expiration time. A timeout of 15 seconds was used for all InformationFinder connections and a maximum of eight threads were used in all threadpools.

In order to evaluate the service response time, a client submitted a series of jobs, waiting with the submission of the next one until the submission of the previous job was completed. Hence, the time measured includes the broker's processing of one single job and all waiting times associated with the resource selection and job submission of that job.

As negotiation of advance reservations is rather time-consuming, tests were performed both with and without reservations. The job response times were grouped into five classes depending on the time required, as shown in tables 1 and 2. The tables summarize five sets of 200 jobs each, showing the average, the minimum and the maximum percentage of job submissions in each interval for each set.

Table 1. Job run time distribution without reservations.

	<2 s	2-5 s	5-8 s	8-11 s	>=11 s
average	9.4%	66.1%	15.4%	3.9%	5.2%
min	7.5%	64%	12%	2%	2.5%
max	12.5%	68.5%	19.5%	7%	8.5%

Table 2. Job run time distribution with reservations.

	<7 s	7-10 s	10-13 s	13-16 s	>=16 s
average	11.4%	37.4%	33.3%	12.4%	5.5%
min	8.5%	32.5%	30%	6.5%	2.5%
max	16.5%	51%	38%	15.5%	10.5%

For jobs submitted without reservations (Table 1), the majority of the submissions take 2–5 seconds, and around 75% of them take less than 5 seconds. This corresponds to jobs where the broker can take advantage of cached resource information and no resource has any exceptionally long response time for job submission. Around 15% of the jobs take 5–8 seconds, corresponding to jobs where the cached information has expired and additional resource queries therefor are performed. The last two categories, i.e., jobs that take more than 8 seconds, include jobs for which additional delays were encountered. These delays were due to an overloaded Grid infrastructure and resulted in two issues: increased time for resource discovery and, more time-consuming, slower dispatch of the job to the selected resource. The latter operation was particularly slow if many jobs recently had been submitted to the same resource.

In the results for submissions performed with advance reservations in Table 2, we basically see the same pattern but with around 5 seconds longer times. One side effect of these longer times is that fewer jobs can benefit from the cached resource information, explaining the smaller number of submissions falling into categories 1–2.

For the throughput tests, jobs were submitted concurrently from many clients, in order to put a high load on the job submission module. To reduce the overhead associated with the first invocation of a Web service, each client submitted a number of jobs. The results show up to 40 successfully handled job submissions per minute, even though the performance varied due to load variations on the Grid infrastructure, resulting in the same anomalies as for the first test. We conclude, however, that for submissions up to at least 40 jobs per minute by the job submission module, it is rather the resources than the broker and job submission service that are the bottleneck.

7 Conclusions

We have presented the design and implementation of a general framework for Grid job submission and resource brokering. The framework is intended to be as middleware independent as possible, and it is therefore to a large extent based on (proposed) Grid and Web services standards. We have demonstrated how the general framework can be integrated in the NorduGrid ARC middleware. In an evaluation of the framework integrated in an environment with ARC resources, we conclude that most jobs can be submitted in less than 5 seconds (less than 13 seconds if using advance reservations), and that the job submission module is capable of achieving a throughput of at least 40 submitted jobs per minute.

The integration requires some effort, in particular when translating the job description language used in ARC back and forth to JSDL, and when converting information retrieved from ARC resources to GLUE. The time spent developing the ARC plugins was however only a fraction of the time required to implement the complete framework, illustrating that implementing plugins for an additional middleware is a feasible method of constructing a feature-rich job submission client for that middleware.

The strive for interoperability often boils down to finding the lowest common denominator between the various systems. In this case, we have been able to adopt the ARC formats to the various standard formats used in our system without losing much of the original ARC features.

Future directions in this work include implementing plugins for additional Grid middlewares. Current efforts focus on support for GT4 and gLite. The future plans also include extending the framework to support coallocation of resources. This will mainly require the development of an alternative submitter module as all the other modules basically have the functionality required.

Acknowledgement

The authors are grateful to Peter Gardfjäll, Aleksandr Konstantinov, and Åke Sandgren. We also acknowledge the three anonymous referees for constructive comments. This work was funded by The Swedish Research Council (VR) under contract 343-2004-953.

References

[1] 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. Internet, 2005. http://infnforge.cnaf.infn.it/docman/view.php/9/90/GLUEInfoModel_1_2_draft_7.pdf.

[2] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web Services Agreement Specification (WS-Agreement). Internet, 2004. https://forge.gridforum.org/projects/graap-wg/document/WS-AgreementSpecification/en/7.

[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. Internet, 2005. https://forge.gridforum.org/projects/jsdl-wg/document/draft-ggf-jsdl-spec/en/21.

[4] Apache Web Services - Axis. http://ws.apache.org/axis.

[5] M. Dalheimer, F.-J. Pfreundt, and P. Merz. Calana: A General-purpose Agent-based Grid Scheduler. Proceedings of Parallel Computing 2005. To be published.

[6] A. Dan, H. Ludwig, and R. Kearney. Cremona: An architecture and library for creation and monitoring of ws-agreements. In *ICSOC'04*, USA, 2004. ACM.

[7] C. Dumitrescu, I. Raicu, and I. Foster. DI-GRUBER: A Distributed Approach to Grid Resource Brokering. SC'05 2005.

[8] P. Eerola, B. Kónya, O. Smirnova, T. Ekelöf, M. Ellert, J. Hansen, J. Nielsen, A. Wäänänen, A. Konstantinov, J. Herrala, M. Tuisku, T. Myklebust, F. Ould-Saada, and B. Vinter. The NorduGrid production Grid infrastructure, status and plans. In *Proc. 4th International Workshop on Grid Computing*, pages 158–165. IEEE CS Press, 2003.

[9] E. Elmroth and J. Tordsson. A Grid resource broker supporting advance reservations and benchmark-based resource selection. In *State-of-the-art in Scientific Computing*, Lecture Notes in Computer Science. Springer-Verlag, 2005.

[10] I. Foster. A Globus toolkit primer. http://www.globus.org/toolkit/docs/4.0/key/GT4_Primer_0.6.pdf.

[11] 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. Internet, 2005. http://www-106.ibm.com/developerworks/library/specification/ws-resource/ws-modelingresources.pdf.

[12] Maui Cluster Scheduler. http://www.clusterresources.com/products/maui.

[13] J. Nabrzyski, J. M. Schopf, and J. Węglarz, editors. *Grid Resource Management*. Kluwer, 2003.

[14] NorduGrid. http://www.nordugrid.org.

[15] D. Ouelhadj, J. Garibaldi, J. MacLaren, R. Sakellariou, and K. Krishnakumar. A multi-agent infrastructure and a service level agreement negotiation protocol for robust scheduling in Grid Computing. In Peter M. A. Sloot et al., editor, *Advances in Grid Computing - EGC 2005*, 2005.

[16] I. Rodero, J. Corbalán, R. M. Badia, and J. Labarta. eNANOS Grid Resource Broker. In Peter M. A. Sloot et al., editors, *Advances in Grid Computing - EGC 2005*, 2005.

[17] W. Smith, I. Foster, and V. Taylor. Scheduling with advanced reservations. In J. Rolim et al., editors, *IPDPS*, 2000.

[18] SweGrid. http://www.swegrid.se.

[19] The Globus Alliance. http://www.globus.org.

[20] S. Venugopal, R. Buyya, and L. Winton. A Grid Service Broker for Scheduling Distributed Data-Oriented Applications on Global Grids. Technical Report GRIDS-TR-2004-1, University of Melbourne, Australia, Feb. 2004.

III

Paper III

A Standards-based Grid Resource Brokering Service Supporting Advance Reservations, Coallocation and Cross-Grid Interoperability*

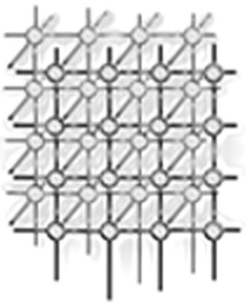
E. Elmroth and J. Tordsson

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

Abstract: The problem of Grid-middleware interoperability is addressed by the design and analysis of a feature-rich, standards-based framework for all-to-all cross-middleware job submission. The architecture is designed with focus on generality and flexibility and builds on extensive use, internally and externally, of (proposed) Web and Grid services standards such as WSRF, JSDL, GLUE, and WS-Agreement. The external use provides the foundation for easy integration into specific middlewares, which is performed by the design of a small set of plugins for each middleware. Currently, plugins are provided for integration into Globus Toolkit 4 and NorduGrid/ARC. The internal use of standard formats facilitates customization of the job submission service by replacement of custom components for performing specific well-defined tasks. Most importantly, this enables the easy replacement of resource selection algorithms by algorithms that addresses the specific needs of a particular Grid environment and job submission scenario. By default, the service implements a decentralized brokering policy, striving to optimize the performance for the individual user by minimizing the response time for each job submitted. The algorithms in our implementation perform resource selection based on performance predictions, and provide support for advance reservations as well as coallocation of multiple resources for coordinated use. The performance of the system is analyzed with focus on overall service throughput (up to over 250 jobs per minute) and individual job submission response time (down to under one second).

Key words: Grid resource broker, standards-based infrastructure, interoperability, advance reservations, coallocation, service-oriented architecture (SOA), Globus Toolkit, NorduGrid/ARC

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2009; **00**:1–38 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*



A standards-based Grid resource brokering service supporting advance reservations, coallocation and cross-Grid interoperability

Erik Elmroth[†] and Johan Tordsson[‡]

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

SUMMARY

The problem of Grid-middleware interoperability is addressed by the design and analysis of a feature-rich, standards-based framework for all-to-all cross-middleware job submission. The architecture is designed with focus on generality and flexibility and builds on extensive use, internally and externally, of (proposed) Web and Grid services standards such as WSRF, JSDL, GLUE, and WS-Agreement. The external use provides the foundation for easy integration into specific middlewares, which is performed by the design of a small set of plugins for each middleware. Currently, plugins are provided for integration into Globus Toolkit 4 and NorduGrid/ARC. The internal use of standard formats facilitates customization of the job submission service by replacement of custom components for performing specific well-defined tasks. Most importantly, this enables the easy replacement of resource selection algorithms by algorithms that addresses the specific needs of a particular Grid environment and job submission scenario. By default, the service implements a decentralized brokering policy, striving to optimize the performance for the individual user by minimizing the response time for each job submitted. The algorithms in our implementation perform resource selection based on performance predictions, and provide support for advance reservations as well as coallocation of multiple resources for coordinated use. The performance of the system is analyzed with focus on overall service throughput (up to over 250 jobs per minute) and individual job submission response time (down to under one second).

KEY WORDS: Grid resource broker; standards-based infrastructure; interoperability; advance reservations; coallocation; service-oriented architecture (SOA); Globus Toolkit; NorduGrid/ARC

[†]E-mail: elmroth@cs.umu.se
[‡]E-mail: tordsson@cs.umu.se
Contract/grant sponsor: Swedish Research Council (VR); contract/grant number: 343-2003-953, 621-2005-3667

Received 14 December 2006
Revised 20 October 2008
Accepted 19 January 2009

1. INTRODUCTION

The emergence of Grid infrastructures facilitates interoperability between heterogeneous resources. Following this development, it is somewhat contradictory that a new level of portability problems has been introduced, namely between different Grid middlewares. Although the reasons are obvious, expected, and almost impossible to circumvent (as the task of defining appropriate standards, models, and best practices must be preceded by basic research and real-world experiments), it makes development of portable Grid applications hard. In practice, the usage of largely different tools and interfaces for basic job management in different middlewares forces application developers to implement custom solutions for each and every middleware. By continued or increased focus on standardization issues, we expect this problem to decrease over time, but we also foresee that it will take long time before the problem can be considered solved, if at all. Hence, we see both a need to more rapidly improve the conditions for Grid application development, and for gaining further experience in designing and building general and standards-based Grid software.

We argue that the conditions for developing portable Grid applications can be drastically improved by providing unified interfaces and robust implementations for a small set of basic job management tools. As a contribution to such a set of job management tools, we here focus on the design, implementation, and analysis of a feature-rich, standards-based tool for resource brokering. This job submission service, designed with focus on generality and flexibility, relies heavily on emerging Grid and Web services standards both for the various formats used to describe resources, jobs, requirements, agreements, etc, and for the implementation of the service itself.

The service is also designed for all-to-all cross-middleware job submission, which means that it takes the input format of any supported middleware and (independently of which input format) submits the jobs to resources running any supported middleware. Currently supported middlewares are the Globus Toolkit 4 (GT4) [30] and NorduGrid/ARC [15]. The service itself is designed in compliance with the Web Services Resource Framework (WSRF) [25] and its implementation is based on GT4 Java WS Core.

The architecture of the service includes a set of general components. To emphasize separation of concerns, each component is designed to perform one specific task in the job submission process. The inter-component interaction is supported by the use of (proposed) standard formats, which increases the flexibility by facilitating the replacement of individual components. The service can be integrated for use with a specific middleware by the implementation of a few minor plugins at well-defined integration points.

The flexible architecture enables partial or complete replacement of the resource selection algorithms with custom implementations. By default, the service uses a decentralized brokering policy, working on behalf of the user [19, 41]. The existing algorithms strive to optimize the performance for an individual user by minimizing the response time for each job submitted. Resource selection is based on resource information as opposed to resource control, and is done regardless of the impact on the overall (Grid-wide) scheduling performance. The resource selection algorithms include performing time predictions for file transfers and a benchmarks-based procedure for predicting the execution time on each resource considered. For enhanced Quality of Service (QoS), the broker also includes features for performing advance resource reservations and coallocation of multiple resources.

The job submission service presented here represents the final version of a second generation job submission tool. For resource allocation algorithms, it partly extends on the algorithms for single job

submissions in the NorduGrid/ARC-specific resource broker presented in [19]. With the development of the second generation tools, principles of SOAs were adapted. Early work on this WSRF-based job submission tool is presented in [18]. The current contribution completes that work and extends it in a number of ways. Hence, one major result of the current article is the completion of the WSRF-based tool into a production quality job submission software.

In summary, our contributions are the following:

- A demonstration of how standard formats and interfaces can be used to support interoperability in terms of all-to-all cross middleware job submission, without restricting functionality to the lowest common denominator. A thorough analysis shows that this can be done with far better performance than required in the envisioned usage scenarios.
- A flexible and portable architecture that allows both customization and replacement of arbitrary components for well-defined subtasks in the the job submission process.
- Resource selection algorithms that can utilize, but do not depend on, sophisticated mechanisms for predicting job and resource performance.
- A standards-based advance reservations framework and its applications in supporting end user QoS.
- Advances to the state-of-the-art in Grid resource coallocation, including the design, implementation, and analysis of an algorithm for arbitrarily coordinated allocations of resources.

The outline of the rest of the paper is organized as follows. The overall system architecture is presented in Section 2. The resource brokering algorithms used in this implementation are described in Section 3, including some further discussions on the intricate issues of resource coallocation and advance reservations. Section 4 illustrates how to design the custom components required to allow job submission to and from additional middlewares, by summarizing the steps required for integration with GT4 and NorduGrid/ARC. The performance of the system is analyzed in Section 5, followed by a presentation of related work in Section 6. Section 7 contains some concluding remarks, including a discussion of our scientific contributions. Information about how to retrieve the presented software is given in Section 8.

2. A STANDARDS-BASED GRID BROKERING ARCHITECTURE

The overall architecture of the job submission and resource brokering service is developed with focus on flexibility and generality at multiple levels. The service itself is made independent of any particular middleware and uses (proposed) standard formats in all interactions with clients, resources, and information systems. It is composed of a set of components that each performs a well-defined task in the overall job submission process. Also in the interaction between these components, (proposed) standard formats are used whenever available and appropriate. This principle increases the overall flexibility and facilitates replacement of individual components by alternative implementations. Moreover, some of these components are themselves designed in a similar way, e.g., making it possible to replace the resource selection algorithm inside the resource brokering component.

The service is implemented using the GT4 (Java WS Core) Web service development framework [24]. This framework combines WSRF functionality with the Axis Web service engine [8]. As the service itself is made independent of any particular middleware, all middleware-specific issues are

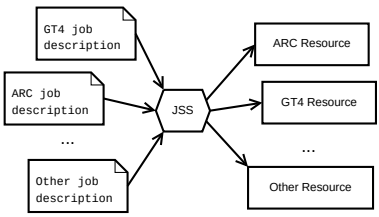


Figure 1. Logical view of interoperability in the job submission service.

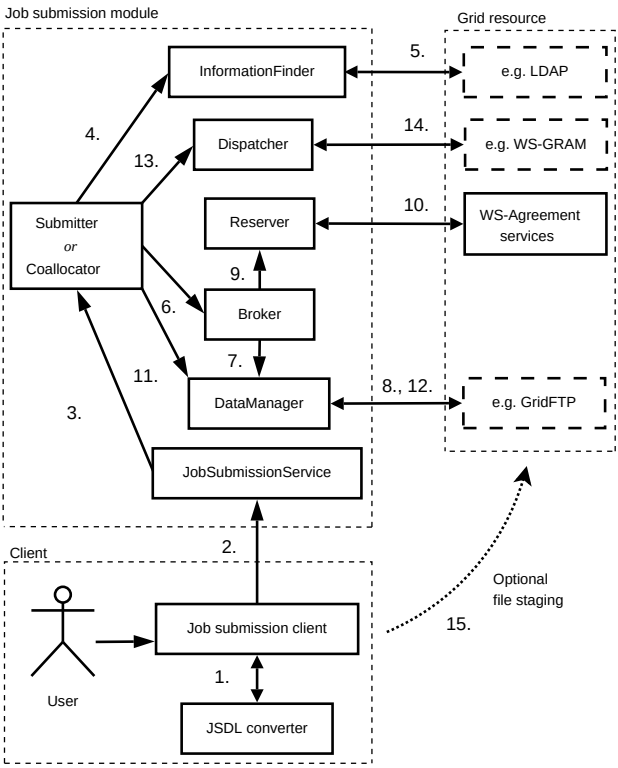
handled by a few, well-defined, plugins. Currently such plugins are available for the GT4 and ARC middlewares. A typical set of middleware plugins constitutes less than ten percent of the general code. Descriptions of the middleware-specific components, including a short discussion about their differences, are found in Section 4.

The service supports job submission to and from any middleware with an X.509 certificate based security framework for which plugins are implemented. This also includes cross-middleware job submission, as illustrated in Figure 1. The figure shows how job requests in the respective job description languages of GT4, ARC or any other supported middleware, are sent to the job submission service (denoted JSS in the figure), which can dispatch the job to a resource that runs any (supported) middleware. This proxy [28] pattern achieves interoperability in the sense of end user transparency, which is in harmony with the ISO definition of interoperability [23]. In contrast to alternative approaches [31, 54], that are based on interoperation through resource side middleware extensions, our solution is non-intrusive as it requires no modifications to the Grid middleware of the resources. However, with our solution, end users must change their job submission software (but can still reuse existing job descriptions), whereas the alternative approaches allow the continued use of middleware-native job submission tools.

In addition to the main job submission and resource brokering service, the framework includes user clients, and an optional advance reservation component that can be installed on the Grid resources for improved QoS. All components are briefly described in sections 2.1–2.3 and their interactions are illustrated in Figure 2.

2.1. Job submission clients

The client module contains two user clients, for standard job submission and for submission of jobs that require coallocation, respectively. The module also includes a plugin for job description translations. An implementation of this plugin converts a job description from the native format specified as input by the user to the Job Submission Description Language (JSDL) [7], a standardized job description format proposed by the Open Grid Forum (OGF) [55]. Users can of course also specify their job requests directly in JSDL.



1. *Job description translation.*
2. Service invocation.
3. Selection of appropriate submission coordinator.
4. Resource discovery and information retrieval.
5. Information system queries and format conversions.
6. Resource brokering.
7. File transfer time estimation.
8. Job data queries.
9. *Advance reservation request.*
10. *Agreement negotiation.*
11. Input file staging.
12. Data transfers.
13. Job submission, including job description translation.
14. Job request.
15. *Client side file staging.*

Figure 2. An architecture overview that shows components, hosts, and information flow. The dashed lines denote different hosts. The boxes show the components, the solid ones are part of the job submission service, whereas the dashed ones illustrate other services it interacts with. A chronological outline of the job submission processes is shown to the right. In this outline, italic font specifies optional tasks.

The job description languages used in various Grid middlewares are very similar in terms of job configuration, e.g., executable to run, arguments, input and output files, execution environment etc., and resource requirements, e.g., number of requested CPUs, required disk space, architecture, available disk, memory, etc. These attributes are straight-forward to translate. However, not all attributes can be translated into JSDL. Each Grid job description language typically contains a few attributes custom to a specific Grid middleware or job submission tool, e.g., instructions to a specific Grid resource broker. Such attributes are not translated in the job submission service as it does not implement all the different scheduling policies of existing Grid resource brokers. Furthermore, attributes specific for one Grid middleware would not be useful when submitting the job to a resource that runs some other middleware. An alternative approach, taken by Kertész et al. [39], is to add all custom attributes as

JSDL extensions. This is a reasonable approach for Kertész et al. as their resource broker forwards job requests to middleware-specific resource brokers, whereas the service described in this paper communicates directly with the resources. However, users of the job submission service can specify custom requirements for their jobs, but through the optional job preferences document described in Section 2.4, instead of through middleware-specific mechanisms. A few finer details about the the translations to JSDL from the job description languages of GT4 and ARC are discussed in sections 4.1 and 4.2, respectively.

The job submission clients can be configured to transfer job input files stored locally at the client host. This file staging is required if the local files neither can be accessed directly by the job submission service nor by the Grid resource.

2.2. Job submission service

The clients send their JSDL job requests to the *JobSubmissionService* that exposes a Web service interface to the functionality offered by the broker, namely submission of a single job or coallocation of a set of jobs. As part of this invocation, the clients delegate user security credentials (i.e., proxy certificates) to the JobSubmissionService, for later use in interaction with resources. The service forwards incoming requests either to the *Submitter* or the *Coallocator*, depending on the type of the request. For each successfully submitted (or coallocated) job, the JobSubmissionService creates a WS-Resource, with information about the job exposed as WS-ResourceProperties. This mechanism for storing state information in Web services is specified by the WSRF [25].

The Submitter (or the Coallocator) coordinates the job submission process, which includes to discover the available Grid resources, gather detailed resource information, select the most suitable resource(s) for the job, and to send the job request to the selected resource. The main difference between the two components is that the Coallocator performs a more complex resource selection and reservation procedure in order allocate multiple resources for coordinated use. The algorithms used by these two components are presented in sections 3.1 and 3.2, respectively.

Resource discovery is handled by the *InformationFinder*. Due to differences in the both communication protocols and Grid information formats used by the various Grid middlewares, the InformationFinder consists of three parts each having a middleware-specific plugin. The *ResourceFinder* contacts a higher level index service to retrieve a list of available Grid resources. Its plugin determines which protocol to use and what query to send to the index service. The *InformationFetcher* queries a single Grid resource for detailed information, such as hardware and software configuration and current load. The *InformationConverter* converts the information retrieved from a Grid resource from the native information format to the format specified by the Grid Laboratory Uniform Environment (GLUE) [4]. The GLUE format defines an information model for describing computational and storage resources in a Grid. To improve the performance of the InformationFinder, threadpools are used for all of these three tasks. For additional performance improvements, the retrieved resource information is cached for a short period of time, which significantly decreases the number of information queries sent to Grid resources during high service load. In order to avoid stale cache entries, resource information is renewed when more recent one becomes available. Metadata included with the resource information specifies how long the retrieved information is valid.

The *Broker* module initially validates incoming job requests, to ensure that a request includes all required attributes, such as the executable to run. Later in the submission process, the Broker is used

by the Submitter (or Coallocator) to rank the resources found by the InformationFinder. The Broker first filters out resources that fail to fulfill the hardware and software requirements specified in the job description, then it ranks the remaining resources after their suitability to execute the job. The resource ranking algorithms may include requesting advance reservations using the *Reserver*, which can create, modify, cancel, and confirm advance reservations using a protocol based on WS-Agreement [5]. The details of the advance reservation protocol and the resource ranking algorithms are given in sections 2.3 and 3.1, respectively. When the ranking is done, the Broker returns a list of the approved resources, ordered by their rank.

The Broker may also use the *DataManager* during resource ranking, a module that performs job submission related data management tasks. This module provides the Broker with estimates of file transfer times, which are predicted from the size and location of each job input and output file. Alternatively, if the Grid middleware supports data replication and/or network performance predictions, the DataManager can use these capabilities to provide better estimates of file transfer times. The DataManager can also stage job input files, unless this task is handled by the client or by the Grid resource executing the job.

The last module used by the Submitter (or Coallocator) is the *Dispatcher*, which sends the job request to the selected resource. As part of this process, the Dispatcher, if required, translates the job description from JSDL to a format understood by the Grid middleware of the resource. The Dispatcher also selects the appropriate mechanism to use to contact the resource. A plugin structure that combines the Chain of Responsibility and Adapter design patterns [28] enables the Dispatcher to perform these tasks without any a priori knowledge of the middleware used by the resource.

2.3. Advance reservations with WS-Agreement

As part of the job submission framework, we have made an implementation of the WS-Agreement specification [5], to be used for negotiating and agreeing on resource reservation contracts. The WS-Agreement module includes an implementation of the AgreementFactory and the Agreement porttypes. However, the Agreement state porttype (which is also part of the WS-Agreement specification) has been left out from the implementation since monitoring the state is not of interest for this type of agreements. It should be remarked that the WS-Agreement implementation itself is completely independent of the service domain (resource reservations) for which it is to be used. We refer to [18] for further details. The WS-Agreement services are the only components that need to be installed on the Grid resource. They enable a client, e.g., the Reserver in the job submission module, to request an advance reservation for a job at the resource.

It should be stressed that it is a priori not known if a reservation can be created on a resource at a given time. The reservation request sent from a client to the AgreementFactory specifies the number of requested CPUs, the requested reservation duration, and the earliest and latest acceptable reservation start times, the latter two forming a window of acceptable start times. Three replies are possible:

1. `<granted>` - request granted.
2. `<rejected>` - request rejected and never possible.
3. `<rejected,T_next>` - request rejected, but may be granted at a later time, `T_next`.

Reply number 1 confirms that a reservation has been successfully created according to the request. Reply number 2 typically indicates that the requested resource does not meet the requirements of the

request, or that the resource rejects the request due to policy reasons. Reply number 3 also indicates a failure, but suggests that a new reservation request, identical to the rejected one, but specifying a later reservation start time (`T_next` or later), may be granted.

An Agreement client can include an optional flag, *flexible*, in the reservation request to specify that the local scheduler may alter the reservation start time within the start time window after the reservation is created. By allowing such a malleable reservation, the local scheduler is given the possibility to rearrange the local schedule. This may improve the resource utilization and partly compensate for the performance penalty imposed by the usage of advance reservations [22].

For advance reservations, two plugin scripts are required at the resource side. The first plugin negotiates reservations with the local scheduler. It is currently implemented for the Maui scheduler [47], but can easily be adopted to work with any local scheduler that supports advance reservations. The second plugin performs admission control of a job that requests to make use of a previously created reservation. This plugin needs to be integrated with the job management mechanism deployed at the Grid resource. The details of the integration of this plugin into existing Grid middlewares are discussed in sections 4.1 and 4.2.

Notably, the job submission service can also handle resources that do not have a reservation capable scheduler and the WS-Agreement services installed, but then, of course, without possibility to make use of the advance reservation feature.

2.4. The optional job preferences document

In addition to the job description, specified either in a middleware-specific format or in JSDL, the job submission framework allows a client to include an optional job preferences document in a job request. For example, this document can be used to choose brokering objective. The user can, e.g., choose between optimizing for an early job start or an early job completion, and can also specify absolute or relative times for the earliest or latest acceptable job start.

The job preferences document may also include information that can improve the brokering decisions, e.g., specification of benchmarks relevant for the application and information about job input and output files. This information is used to improve the resource selection process as described in Section 3.1. Just as it is optional to provide the job preferences document itself, all parameters in the document are optional.

3. ALGORITHMS FOR RESOURCE (CO)ALLOCATION

The general problem of resource brokering is complex, and the design of algorithms is highly dependent on the scheduling objectives, the type of jobs considered, the users' understanding of the application requirements, etc. For a general introduction to these issues, see e.g., [62, 76].

In order to facilitate the use of custom brokering algorithms, the job submission service architecture presented in Section 2 is designed for easy modification or replacement of brokering algorithms. The predefined algorithms provided for single job submission and for submission of jobs requiring resource coallocation are presented below in sections 3.1 and 3.2, respectively.

3.1. Resource ranking algorithms

The algorithm for submitting single jobs, implemented in the Submitter module, strives to identify the resource that best fulfills the brokering objective selected by the user. The two alternative brokering objectives are to find the resource that gives the earliest job completion time or the one the gives the earliest job start time. Notably, these two objectives give the same result if all resources are identical and jobs do not transfer any output data. This is however an unlikely scenario in Grid environments.

In order to identify the most suitable resource, the Broker makes a prediction of either the *Total Time to Delivery* (TTD) or the *Total Time to Start* (TTS), respectively, using two different *Selectors*. These predictions are based on time estimation algorithms originally presented in [19]. In order to estimate the TTS, the broker needs to, for each resource considered, predict the time required for staging of the executable and the input files, and the time the job must wait in the batch queue. In addition, the estimation of the TTD also requires predictions of the job execution time and of the time required for staging the output files. The time predictions for these four tasks are performed by four different *Predictors*. Notably, by modifying the Selectors and/or Predictors, or by defining new ones, customization of the brokering algorithm is rather straightforward. The existing Predictions have basic functionality as follows.

Time predictions for file staging. If the DataManager provides support for predicting network bandwidth, for resolving physical locations of replicated files, or for determining file sizes (see Section 2), these features are used by the stage in predictor for estimating file transfer times. If no such support is available, e.g., depending on the information provided by the Grid middleware used, the predictor makes use of the file transfer times optionally provided by the user. Notably, the time estimate for stage in is important not only for predicting the TTD but also for coordinating the start of the execution with the arrival of the executable and the input files if an advance reservation is created for the job. The stage out predictor only considers the optional input provided by the user, as it is impossible to predict the size of the job output and hence also the stage out time without user input.

Time predictions for batch queue waiting. The most accurate prediction of the batch queue waiting time is obtained by using advance reservations, which gives a guaranteed job start time. If the resource does not support advance reservations or the user chooses not to activate this feature, a less accurate prediction is made from the information provided by the resource about current load. This rough estimate does however not take into account the actual scheduling algorithm used by the batch system.

Time predictions for application execution. The prediction of the time required for the actual job execution is performed through a benchmark-based estimation that takes into account both the performance of the resources and the characteristics of the application. This estimation requires that the user provides the following information for one or more benchmarks with performance characteristics similar to that of the application: the name of the benchmark, the benchmark performance for some system, and the application's (predicted) execution time on that system. Using this information, the application's execution time is estimated on other resources assuming that the performance of the application is proportional to that of the benchmark. If multiple benchmarks are specified by the user, this procedure is repeated for all benchmarks and then the average result is used as a prediction of the execution time. In order to handle situations where resources do not provide all requested benchmarks, the prediction algorithm includes a customizable procedure for making conservative predictions. We remark that it is the responsibility of the user to ensure that performance characteristics of the selected benchmarks are representative for the application. A benchmark need however not at all be formal

or well-established. It may equally well be a performance number of the actual application code for some predefined problem. The requirement for an (estimate of the) application execution time should furthermore be easy to fulfill as users typically submit the same application multiple times.

Grid application runtime prediction consists of two separate problems. The first is to predict the performance of an application with fixed parameters on a set of machines, given the performance for the application (with the same set of parameters) on a known machine. The second problem is to, on a known machine, estimate the performance of an application for varying parameters based on knowledge of the performance of the application for a given set of parameters. The TTD and TTS algorithms address the first problem. The second problem is not specific to Grid jobs and have been studied extensively [21, 53, 64].

The TTD (and TTS) application performance models can be used to accurately predict the behavior of a wide range of applications, including compute and/or data intensive jobs. By considering input and output data transfers, resource access wait time and actual application execution, these performance models take into account and combine previous models such as *LeastLoaded* [59] and *DataPresent* [59] as well as the application-specific computational performance of the resource.

The implementations of the TTD and TTS models are carefully designed to predict higher performance for resources for which more detailed information and more accurate performance estimation mechanisms are available. From a resource selection perspective, it is however important to use the same metric (TTD or TTS) for comparison of all the resources of interest. This necessitates the use of coarse-grained prediction methods, e.g., estimating the queue waiting time from current resource load, when no better mechanism is available.

3.2. Coallocation

A coallocation mechanism is required in order to start a Grid job that makes coordinated use of more than one resource. The algorithm used for performing coallocation is implemented in the Coallocator module (see Section 2), which makes use of the same underlying components as the Submitter that allocates resources for single jobs.

The coallocation algorithm presented here share some characteristics with an algorithm by Mateescu [46]. Both these algorithms perform coallocation in an *on-line* style, i.e., the set of resources to coallocate is determined (and reserved) incrementally. The algorithm described in this paper is based on the advance reservation protocol outlined in Section 2.3 and does hence assume that it not known a priori whether a resource can be reserved at a given time or not. An alternative coallocation algorithm suggested by Wäldrich et al. [73] can be classified as *off-line* coallocation. This algorithm assumes preknowledge of when resources are available for reservation, determines the set of resources to use off-line, and creates reservations once a suitable set is found. A more in-depth discussion of the benefits and drawbacks of the respective approaches can be found in Section 6.

The main algorithm for identifying and allocating suitable resources for coordinated use is described in Section 3.2.3. The presentation of the overall algorithm is preceded by a more precise definition of the coallocation problem in Section 3.2.1, and an overview of the main ideas used in the algorithm in Section 3.2.2. In Section 3.2.4, the algorithm is illustrated by a coallocation scenario that highlights some of its key features. This is followed by a discussion of some of the more intricate parts of the algorithm.

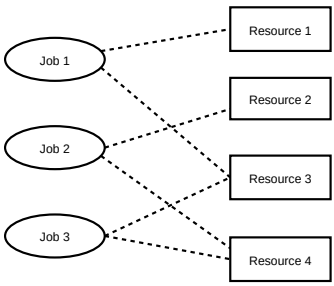


Figure 3. Subjobs and their possible resources viewed as a bipartite graph.

3.2.1. Problem definition

The input to the (on-line) coallocation problem is the following:

1. A set of $n \geq 2$ job requests: Jobs $= \{J_1, J_2, \ldots, J_n\}$.
2. A set of n resource sets, where each of the n sets contains the resources that are identified to have the capabilities required for one of the jobs:
   Resources $= \{R_1, R_2, \ldots, R_n\}$ where $R_1 = \{R_{11}, R_{12}, \ldots, R_{1m_1}\}, R_2 = \{R_{21}, R_{22}, \ldots R_{2m_2}\}$, $\ldots, R_n = \{R_{n1}, R_{n2}, \ldots, R_{nm_n}\}$, $|R_i| = m_i$, are the resources that can be used by $J_1, J_2, \ldots, J_n$, respectively, Notably, the same resource may appear in more than one R_i.

A coallocated job requires a matching $\{J_1 \rightarrow R_{1j_1}, J_2 \rightarrow R_{2j_2}, \ldots, J_n \rightarrow R_{nj_n}\}, J_i \in$ Jobs, $R_{ij_i} \in R_i, 1 \leq i \leq n, 1 \leq j_i \leq m_i$ such that J_i has a reservation at resource R_{ij_i}.

For clarity, the coallocation algorithms are described for the case when all reservations start simultaneously. The algorithms can however perform any kind of job start time coordination, by allowing individual job start time offsets from a common start time. Although currently only implemented for computational jobs, the coallocation algorithms are general enough to allow coallocation of other resource types, e.g., networks. The only requirement for a resource type to be coallocated is the existence of an advance reservation mechanism that supports the protocol described in Section 2.3. The term *job* in the following descriptions can hence be read as *request for resource* (that supports advance reservations).

The jobs and resources forming the input to the coallocation problem can be expressed as a bipartite graph as illustrated in Figure 3. An edge between a job and a resource in the graph represents that the resource has the capabilities required to execute the job. The problem of pairing jobs with resources (by reserving the resources for the jobs) can hence be viewed as a bipartite graph matching problem. A matched edge in the graph of jobs and resources represents that a reservation for the job is created at the resource. In this context, a coallocated job is a complete matching of the jobs to some set of resources. We note that some resources can execute (or hold reservations for) multiple jobs concurrently, and can hence be matched with more than one job.

3.2.2. Algorithm overview

In overview, the coallocation algorithm strives to find the earliest common start time for all jobs within a job start window $[T_e, T_l]$, where T_e and T_l are the earliest and latest job start time the user accepts. The earliest common job start time is achieved by the creation of a set of simultaneously starting reservations, one for each job. For practical reasons, a somewhat relaxed notion of simultaneous job start is used, reducing the simultaneous start time constraint to that all jobs must start within the same (short) period of time, expressed as a time window $[t_e, t_l]$. We remark that the coallocation requires the clocks of all resources to be synchronized in the order of $[t_e, t_l]$, using e.g., the Network Time Protocol (NTP) [49]. Typical values of $|[t_e, t_l]|$ is in the order of a half minute to a few minutes, whereas NTP can keep clocks synchronized within milliseconds [49]. Even though poorly synchronized clocks do not cause the coallocation algorithm itself to fail, clock drifts delay the start of the coallocated job, which in turn may cause batch system preemption as the total execution time of the job is increased.

The coallocation algorithm operates in iterations. Before the first iteration, the $[t_e, t_l]$ window is aligned at the start of the larger $[T_e, T_l]$ window. In each iteration, reservations starting simultaneously, i.e., within the start time window $[t_e, t_l]$, are created for each job. Alternatively, previously created reservations are modified (moved to a new start time window), or reservations are exchanged between jobs, all to ensure that each job gets a reservation starting within the $[t_e, t_l]$ window. The exchange of reservations is performed to increase the number of matched jobs when a critical resource is already reserved for a job that may use alternative resources. If no reservation can be created for some job during the $[t_e, t_l]$ window, this window is moved to a later time and the algorithm starts a new iteration. This sliding-window process is repeated with additional iterations either until each job has a reservation (success) or the earliest possible reservation starts too late (failure).

3.2.3. Coallocation algorithms

The main coallocation algorithm is given in Algorithm 1 and the procedure (based on *augmenting paths*) for exchanging reservations between jobs is described in Algorithm 2. Further motivation for the most important steps of the algorithms and a discussion of their more intricate details are found in Section 3.3.

The inputs to Algorithm 1 are the set of jobs and the sets of resources capable of executing the jobs as defined in Section 3.2.1. Additional inputs are the $[T_e, T_l]$ window specifying the acceptable start time interval and ϵ, the maximum allowed job start time deviation.

In Step 1 of the algorithm, the currently considered start time window $[t_e, t_l]$ is aligned to the start of the acceptable start time interval. This $[t_e, t_l]$ window is moved in each iteration of the algorithm, but its size is always ϵ. The main loop, starting at Step 2 is repeated until either all jobs have a reservation within the $[t_e, t_l]$ window (success) or the $[t_e, t_l]$ window is moved outside $[T_e, T_l]$ (failure). In Step 3 of the algorithm, an initially empty set A is created for jobs for which it is neither possible to create a new reservation starting within $[t_e, t_l]$, nor to modify an existing reservation to start within this window. Step 4 of the algorithm defines T_{best}, where in time to align the $[t_e, t_l]$ window if an additional iteration of the main loop should be required. To ensure termination of the main loop in the case when all reservation requests fail, and no reservation ever will be possible (reply number 2 in the reservation protocol), T_{best} is set to infinity. This variable is assigned a finite value in steps 12 and 18 if any failed reservation request returns a next possible start time (reply number 3 in the reservation protocol).

Algorithm 1 Coallocation

Require: A set of $n \geq 2$ resource requests (job requests) Jobs $= \{J_1, J_2, \ldots, J_n\}$.
Require: A set of resources with the capabilities required to fulfill these requests. Resources $= \{R_1, R_2, \ldots, R_n\}$ where $R_1 = \{R_{11}, R_{12}, \ldots, R_{1m_1}\}$, $R_2 = \{R_{21}, R_{22}, \ldots R_{2m_2}\}$, $\ldots R_n = \{R_{n1}, R_{n2}, \ldots, R_{nm_n}\}$ are the resources that can be used by $J_1, J_2, \ldots, J_n$, respectively.
Require: A start time window $[T_e, T_l]$ specifying earliest and latest acceptable job start.
Require: A maximum allowed start time deviation ϵ.
Ensure: A set n of simultaneously starting reservations, one for each job in Jobs.

1: Let $t_e \leftarrow T_e$ and $t_l \leftarrow T_e + \epsilon$.
2: **repeat**
3: Let $A \leftarrow \emptyset$ be the set of jobs for which path augmentation should be performed.
4: Let $T_{\text{best}} \leftarrow \infty$ be the earliest time later than $[t_e, t_l]$ that some reservation can start.
5: **for** each job $J_i \in$ Jobs, $1 \leq i \leq n$, that does not have a reservation starting within $[t_e, t_l]$ **do**
6: **if** J_i already has a reservation starting outside (before) $[t_e, t_l]$ **then**
7: Modify the existing reservation to start within $[t_e, t_l]$.
8: **if** Step 7 fails, or if J_i had no reservation **then**
9: Create a new reservation starting within $[t_e, t_l]$ for J_i at some $r \in R_i$.
10: **if** Step 9 fails **then**
11: Add J_i to A.
12: Let $T_{\text{best}} \leftarrow \min\{T_{\text{best}}$, the earliest `T_next` value returned from Step 9$\}$.
13: **if** each job $J \in A$ may be augmented **then**
14: **for** each job $J \in A$ **do**
15: Find an augmenting path P starting at J using breadth-first search.
16: Update reservations along the path P using Algorithm 2.
17: **if** Step 16 fails **then**
18: Let $T_{\text{best}} \leftarrow \min\{T_{\text{best}}$, the earliest `T_next` value returned from Step 16$\}$.
19: **if** some job in J has no reservation starting within $[t_e, t_l]$ **then**
20: Let $t_l \leftarrow T_{\text{best}}$ and $t_e \leftarrow (t_l - \epsilon)$.
21: **if** $t_e > T_l$ **then**
22: The algorithm fails.
23: **until** all jobs have a reservation starting within $[t_e, t_l]$
24: Return the current set of reservations.

Step 5 is performed for each job that has no reservation within the $[t_e, t_l]$ window. This applies to all jobs unless the window has been moved less than ϵ since the last iteration, in which case some previously created reservations still may be valid. In Step 6, it is tested whether the job already has a reservation from a previous iteration, that starts too early for the current $[t_e, t_l]$ window. If so, this reservation is modified in Step 7 by requesting a later start time (within the new $[t_e, t_l]$ window). The condition in Step 8 ensures that Step 9 is only executed for jobs that have no reservation, either because no reservation could be created in the previous iteration of the algorithm or because the job has lost its reservation due to a reservation modification failure. In Step 9, a new reservation is created for the job by first trying to reserve the resource highest ranked by the broker, and upon failure retry with the second highest ranked resource etc., until either a reservation is created or all requests have failed.

In case one or more reservation requests in Step 9 receive reply number 3 in the reservation protocol ("<rejected, T_next>") the earliest of these T_next values is stored for usage in Step 12.

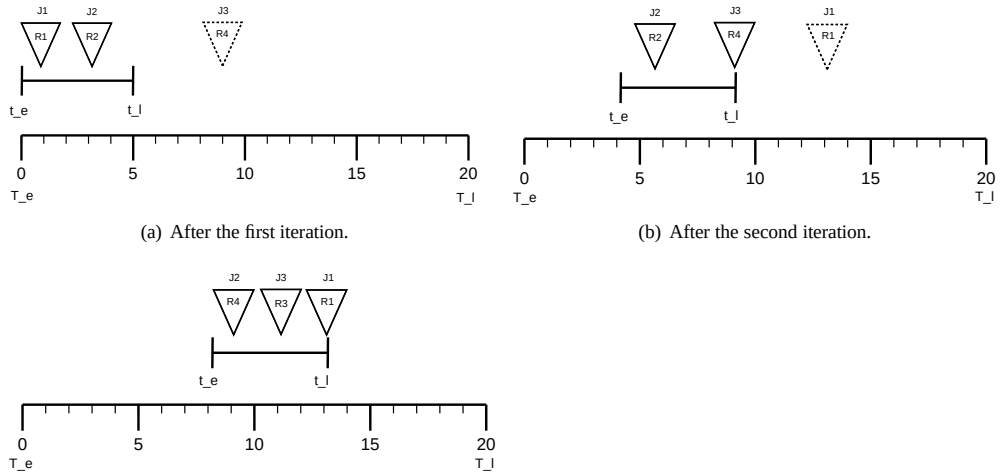
Step 10 tests if all reservation requests have failed for a job J_i, and if so, this job is included in A in Step 11 to be considered for path augmentation later. Step 12 updates T_{best} with the earliest T_next value obtained in Step 9, if such a value exists. In Step 13, it is tested whether augmentation can be used for each job in A. This test is done for a job J by ensuring that some other job J' holds a reservation for a resource that J can use. If no such other job J' exists, there is no reservation to modify to suit job J and no augmenting path can hence be found. As the goal of the algorithm is to match all jobs, Step 13 ensures that all jobs in A are eligible for augmentation. It is of no use if the current matching can be extended with some, but not all, unmatched jobs.

If augmentation techniques can be used according to the test in Step 13, the loop in Step 14 is executed for each job in A. In Step 15, an augmenting path of alternating unmatched and matched edges, starting and ending in an unmatched edge, is found using breadth-first search. In Step 16, the reservations (matchings) along this augmented path are updated using Algorithm 2. The path updating algorithm includes both modifications of existing reservations and creation of new ones. If any of these operations fail and return reservation request reply number 3, the earliest T_next value is, in analogy with Step 9, stored for usage in Step 18. Step 17 tests if the update algorithm failed, and if so, Step 18 updates T_{best}. Step 19 tests whether the main coallocation algorithm will terminate, or if another iteration is required. In the latter case, the $[t_e, t_l]$ window is updated in Step 20. In order to move the window as little as possible, i.e., to ensure the earliest possible job start, t_l is set to T_{best} and t_e is updated accordingly. Step 21 ensures that the $[t_e, t_l]$ window has not moved beyond the $[T_e, T_l]$ window. If this is the case, the algorithm fails (Step 22). Once the loop in Step 2 terminates without failure, the coallocation algorithm is successful and the current set of reservations is returned (Step 24).

Algorithm 2 Update augmenting path

Require: An augmenting path $P = \{J_1, R_1, \ldots, J_n, R_n\}, n \geq 2$ where R_i is reserved for J_{i+1}.
Ensure: An augmenting path $P = \{J_1, R_1, \ldots, J_n, R_n\}, n \geq 2$ where R_i is reserved for J_i.
1: Create a new reservation for J_n at R_n.
2: **if** Step 1 fails **then**
3: The algorithm fails.
4: **for** $i \leftarrow (n-1)$ downto 1 **do**
5: Modify the existing reservation at resource R_i for job J_{i+1} to suit job J_i.
6: **if** the modification in Step 5 fails **then**
7: The algorithm fails.
8: Return.

Algorithm 2 is invoked in Step 16 of Algorithm 1 to modify the reservations along an augmented path. In Step 1 of Algorithm 2, a new reservation is created for job J_n, as the existing reservation for this job will be used by job J_{n-1}. If the creation of the new reservation fails, it is of no use to modify the existing reservations, and the algorithm fails (Step 3). The loop in Step 4 is performed for all existing reservations. In Step 5, the reservation currently created for job J_{i+1} is modified to suit the requirements of job J_i. This modification typically includes changing the number of reserved CPUs and the duration of the reservation, but the reservation start time is never changed. Step 6 tests



(a) After the first iteration.

(b) After the second iteration.

(c) After the third and final iteration.

Figure 4. Example execution of the coallocation algorithm.

if the modification fails. If so, it is not meaningful to continue the execution and Step 7 terminates the algorithm (with failure). Once the loop in Step 4 terminates without error and Step 8 is reached, the algorithm is successful.

Algorithms 1 and 2 describe the simplified coallocation scenario where all subjobs are coallocated for a concurrent job start. The actual implemented algorithm is more general as it allows each subjob start time to have an arbitrary offset from a common start time. This general scenario requires two minor extensions to the described algorithms. First, all comparisons with the $[t_e, t_l]$ window (steps 5, 6, 7, 9, 19, and 23 of Algorithm 1) must take into account the individual subjob's offset from this window. Secondly, special care must be taken when exchanging reservations between subjobs, as these need not have a common start time. For clarity, these extensions are left out from the descriptions of algorithms 1 and 2.

3.2.4. Example execution

The following example illustrates the execution of the coallocation algorithm. Let the input to the algorithm be Jobs $= \{J_1, J_2, J_3\}$ and Resources $= \{\{R_1, R_3\}, \{R_2, R_4\}, \{R_3, R_4\}\}$. This scenario corresponds to the bipartite graph shown in Figure 3. Let the start time window $[T_e, T_l]$ be $[0900, 0920]$ (20 minutes) and let the maximum allowed start time deviation, ϵ, be 5 minutes. Notably, we have for clarity kept $[T_e, T_l]$ rather small. In practice, its size may vary from a few minutes to several hours or even days. The size ϵ of the small start time window $[t_e, t_l]$ is however typically only a few minutes.

In the first iteration of the algorithm, a reservation is created for J_1 at R_1 and one for J_2 at R_2. These reservations are shown as solid triangles in Figure 4(a). However, no reservation starting early enough can be created for J_3. The earliest possible reservation (at R_4), which would start a few minutes too late, is shown as a dashed triangle in Figure 4(a) . Path augmentation techniques cannot be used for J_3 as there is no other job holding a reservation for a resource that J_3 can use, i.e., neither J_1 nor J_2 has a reservation at R_3 or R_4, which are the only resources that meet the requirements of J_3.

In next iteration, the $[t_e, t_l]$ window is moved and aligned with the earliest possible reservation start for J_3. A reservation for J_3 is created at R_4. The reservation for J_2 at R_2 is modified to start within the new $[t_e, t_l]$ window. These two reservations are represented by the solid triangles in Figure 4(b). The reservation for J_1 at R_1 can however not be moved to within $[t_e, t_l]$, and is hence implicitly cancelled. Furthermore, no new reservation can be created for J_1 within $[t_e, t_l]$. Path augmentation techniques cannot be used to create an additional reservation as neither J_2 nor J_3 has reserved one of R_1 and R_3. The earliest possible reservation for J_1 (at R_1) is shown as a dashed triangle in Figure 4(b).

In the next iteration, t_l is set to the earliest possible start of J_1 and t_e is adjusted accordingly. The reservation that in the previous iteration was possible for J_1 at R_1 is created, illustrated by a solid triangle in Figure 4(c). The reservation for J_3 at R_4 already starts within $[t_e, t_l]$ and requires no modification. For job J_2 the existing reservation cannot be moved to within $[t_e, t_l]$ and is hence cancelled. It is furthermore not possible to create a new reservation for J_2. The path augmentation algorithm can however be applied. Starting from J_2 in the bipartite graph in Figure 3, a breadth-first search is performed according to Step 15 of Algorithm 1. This search finds a resource that J_2 can use (R_4), which is currently reserved by another job (J_3), which in turn can use another resource (R_3). The resulting augmenting path is $\{J_2, R_4, J_3, R_3\}$. Next, Algorithm 2 is invoked with this path as input. The algorithm creates a new reservation for J_3 at R_3, and modifies the existing reservation for J_3 (at R_4) to suit J_2. The resulting reservations (for J_2 and J_3) are shown as solid triangles in Figure 4(c). Since each job has a reservation starting within $[t_e, t_l]$ (and inside $[T_e, T_l]$), the coallocation algorithm terminates and the coallocation request is successful.

3.3. Discussion of Quality of Service issues

We here discuss advance reservations and the properties of the bipartite matching algorithm in more detail and also motivate the usage of path augmentation techniques.

3.3.1. Regarding the use of advance reservations and coallocation

It should be remarked that how and to what extent advance reservations should be used, partly depends on the Grid environment. The current algorithms are designed for use in medium-sized Grids, and with usage patterns where the majority of the jobs do not request advance reservations. In Grids where hundreds or even thousands of resources are suitable candidates for a user's job requests, the algorithms requesting advance reservations should be modified to first select a subset of the resources before requesting the reservations. In order to allow a majority of the Grid jobs to make use of advance reservations, it is probably necessary to have support for, and make effective usage of, the "flexible" flag (see Section 2.3) in all local schedulers, in order to maintain an efficient utilization of the resources.

3.3.2. *Modifications of advance reservations*

The coallocation algorithm modifies existing reservations as if the modify operation is atomic, even though the current implementation actually first releases the existing reservation and then creates a new one. The reason for this is that the Maui scheduler [47], one of the few batch system schedulers that support advance reservations, has no mechanism to modify an existing reservation. This means that the modification operation, in unfortunate situations, may lose the original reservation even if the new one could not be created. This occurs when the scheduler decides to use the released capacity for some other job before it can be reclaimed.

We also remark that the WS-Agreement specification [5] does not include an operation for renegotiation of an existing agreement (reservation). A protocol for managing advance reservations, including atomic modifications of existing reservations is discussed in [61]. To the best of our knowledge, there exists neither an implementation of this protocol nor a local scheduler with the reservation mechanisms required to implement it. Atomic reservation modifications may very well be included in future versions of the WS-Agreement standard (or defined by a higher level service, such as the currently immature WS-AgreementNegotiation [6]) and supported by new releases of batch system schedulers. Should this happen, the coallocation algorithm itself needs no modification, and it furthermore becomes more efficient, as failed reservation modifications causes extra iterations of the algorithm to be executed.

3.3.3. *Properties of the bipartite matching algorithm*

In the bipartite graph representing jobs and resources, an edge between a job and a resource represents that the resource has the capabilities required to execute the job. We can however not know a priori that the resource actually can be reserved for the job at the time requested. Seen from a graph theoretic perspective, it is not certain that the edges in this bipartite graph actually exist (e.g., at a particular time) before we try to use them in a matching. Given the above facts, it is not possible to completely solve the coallocation problem using a bipartite matching algorithm that precalculates the matching off-line. Therefore, we use a matching algorithm that (on-line) gradually increases the size of the current matching (initially containing no matched edges at all), and use path augmentation techniques to resolve conflicts. A more in-depth discussion of the difference between on-line and off-line coallocation algorithms is found in Section 6.

3.3.4. *Path augmentation considerations*

Path augmentation techniques are used when the coallocation algorithm fails to reserve a resource for a job, but it is possible that this situation can be solved by moving some other reservation (for the same coallocated job) to another resource. In order to reduce the need for path augmentation, we strive to allocate resources in decreasing order of the "size" of their requirements, with the size defined in terms of the requested number of CPUs, required memory, and requested job runtime. We however remark that it is in the general case not possible to perfectly define such an ordering. For example, if one job requires two hours run time and one GB memory, and another only one hour but requires two GBs memory, it is not obvious which of these jobs to first create a reservation for.

The usage of breadth-first search when finding augmented paths guarantees that the shortest possible augmented path is found. Both the initial sorting of the job list and the usage of breadth-first search reduces the number of reservation modifications. This both improves the performance of the algorithm as the updating of an augmented path is time-consuming (see Section 5 for more details), and reduces the risk of failures that occur due to the non-atomicity of the update operation as described in Section 3.3.2.

It should also be noted that the test performed in Step 13 of Algorithm 1 may cause false positives, as it is assumed that path augmentation is possible before actually performing the advance reservations required to augment the path. However, no false negatives are possible, i.e., if the test fails to find a job J' with a reservation that can be used to by job J, then no augmenting path exists.

4. CONFIGURATION AND MIDDLEWARE INTEGRATION

This section discusses how to configure the job submission service, with focus on the middleware integration points. We illustrate the integration by describing the custom components required for using the job submission service with two Grid middlewares, GT4 and ARC. In addition, the configuration of the WS-Agreement services is briefly covered.

Integration of a Grid middleware in the job submission service is handled through the service configuration. This configuration determines which plugin(s) to use for each middleware integration point. Note that the job submission service can have multiple plugins for the same task, enabling it to simultaneously communicate with resources running different Grid middlewares. Using the chain-of-responsibility design pattern, the plugins are tried, one after another, until one plugin succeeds in performing the current task. The configuration file specifies which plugins to use in the InformationFinder and the Dispatcher. This file also determines connection timeouts, the number of threads to use in the threadpools, and default index services. The client is configured in a separate file, allowing multiple users to share a job submission service while customizing their personal clients. The client configuration file determines which job description translator plugins to use, and also specifies some settings related to client-side file staging.

The configuration of the WS-Agreement services determines which *DecisionMaker(s)* to use. A DecisionMaker is a plugin that grants (or denies) agreement offers of a certain agreement type. A DecisionMaker uses two plugin scripts to perform the actions required to create and destroy agreements. For the advance reservation scenario, these plugin scripts interacts with the local scheduler in order to request and release reservations.

4.1. Integration with Globus Toolkit 4

The GT4 middleware does, among other things, provide Web service interfaces for fundamental Grid tasks such as job submission (WS-GRAM), monitoring and discovery (WS-MDS), and, data transfer (RFT) [24]. The job submission client plugin for GT4 job description translation is straightforward. The only issue encountered is that job input and output files are specified using the same attribute in JSDL, whereas the GT4 job description format uses two different attributes for this.

There is no fixed information hierarchy in GT4, any type of information can be propagated between a pair of WS-MDS *index services*. A basic setup (also used in our test environment) is to have one

index service per cluster, publishing information about the cluster, and one additional index service that aggregates information from the other ones. Thus, the typical GT4 information hierarchy does not really fit the infrastructure envisioned by the job submission service, with one or more index servers storing (only) contact information to clusters. However, by using an XPath query in the GT4 ResourceFinder plugin, it is possible to limit the information returned from the top level index service to cluster contact information only. This list of cluster addresses is sent to the GT4 InformationFetcher plugin, that (also using XPath) queries each resource in more detail. Both these plugins communicate with the Grid resource using Web service invocations. The InformationConverter plugin for GT4 is trivial as resource information in GT4 is described in the GLUE format.

The GT4 Dispatcher plugin converts the job description from JSDL to the job description format used in GT4 and next sends the job request to the GT4 WS-GRAM service running on the resource by invoking the job request operation of the service. This procedure becomes more complicated if the Grid resource is to stage (non-local) job input files, in which case the user's credentials must be delegated from the GT4 Dispatcher plugin to the resource.

The WS-Agreement services themselves require no middleware-specific configuration. However, job requests that claim a reservation must be authorized, i.e., it must be established that the user requesting the job is the same as the one that previously created the reservation. This is done by comparing the distinguish names in the X.509 certificate credentials used for the two tasks. In GT4, an Axis request flow chain that intercepts the job request performs this test. Authorization is hence performed in two steps, first by the custom Axis flow chain, and then by the GT4 gridmap authorization mechanism used by WS-GRAM.

4.2. Integration with ARC

The ARC middleware is based on Globus Toolkit 2 (GT2), but replaces some GT2 components, including the GRAM which is substituted by a *GridFTP server* that accepts job requests and a *Grid Manager* that manages accepted Grid jobs through their execution.

The information system in ARC uses GT2 tools, and is organized in a hierarchy where a GIIS server keeps a list of available GRIS (and GIIS) servers, which periodically announce themselves to the GIIS. Another configuration, used in some ARC installations, is to aggregate all GRIS information in the GIIS. The ARC ResourceFinder and InformationFetcher plugins use LDAP to retrieve lists of available resources and detailed resource information, from the GIIS and GRIS respectively. The resource information is described using an ARC-specific schema, and must hence be translated to the GLUE format by the ARC InformationConverter plugin. The ARC and GLUE information models are not fully compatible, but most attributes relevant to resource brokering, e.g., hardware configuration and current load, can be translated between the two models.

The ARC Dispatcher plugins converts the JSDL job description to the GT2 RSL-style format used in ARC and sends the resulting job description to the ARC GridFTP server, i.e., the Dispatcher plugin is a GridFTP client.

Authorization of job requests claiming a reservation is done similarly as in GT4 (by comparing distinguished names). A plugin structure in the ARC Grid Manager enables interception of the job request at a few predefined steps. One such plugin performs the reservation authorization before the job is sent to the local batch queue.

5. PERFORMANCE EVALUATION

The following section evaluates the performance of the job submission service, with an exclusive focus on the service itself, i.e., evaluating how it performs under varying configurations and load.

The resource selection algorithms described in this contribution are based on an estimate of the time required to execute a job in the Grid, either the TTS or the TTD. The accuracy of the algorithms varies with the accuracy of the predictions, which in turn depends both on the mechanisms available, e.g., advance reservations; and input provided by the user, e.g., relevant benchmarks and file transfer time estimates. Given good enough user input and prediction mechanisms as described above, the resource selection algorithms can give arbitrarily good predictions. For this reason, no evaluation of their performance is included in the paper.

There are several factors that affect the performance of the job submission service, including the Grid middleware deployed on the resources, the number of resources, the local scheduler used by the resources, and whether advance reservations are used or not. In order to evaluate these factors, the performance analysis includes measuring, for varying load, (1) the response time, i.e., the time required for a client to submit a job, and (2) the service throughput, i.e., the number of jobs submitted per minute by the service. In addition, the performance of the coallocation algorithm is analyzed.

5.1. Background and test setup

The performance of the job submission service is evaluated using the DiPerF framework [13]. DiPerF can be used to test various aspects of service performance, including throughput, load and response time. A DiPerF test environment consists of one *controller* host, coordinating and collecting output from a set of *testers* (clients). All testers send requests to the service to be tested and report the measured response times back to the controller. Each tester runs for a fixed period of time, and invokes the service as many times it can (in this case, submits as many jobs as possible) during the test period.

The response time measured by a tester includes the time required to establish secure connections to the job submission service, to delegate the user's credential to the service, and to submit the job. The response time also includes time for service side tasks such as broker job processing and interactions with index servers and resources. The throughput is computed in DiPerF by counting the number of requests served during each minute. This calculation is done off-line when all testers have finished executing.

GT4 clients developed using Java have an initial overhead in the order of seconds due to the large number of libraries loaded upon start up, affecting the performance of the first job submitted by each client. As a result, a simple request-response Web service call takes approximately five seconds using a Java client (subsequent calls from the same client are however much faster), whereas a similar call takes less than half a second for a corresponding C client. To overcome this obstacle, a basic C job submission client is used in the performance tests.

The evaluation is performed with resources running either GT4 or ARC. In order to better understand how eventual bottlenecks in the Grid middleware effect the performance of the job submission service, each test uses a single Grid middleware on the resource side. We expect cross-middleware submission and resource brokering in mixed middleware Grids to be slightly slower, as the broker encounters the union of all bottlenecks in the used middlewares in such a scenario.

The performance measurements have been performed in a test environment with four small clusters, each equipped with a 2 GHz AMD Opteron CPU and 2 GB memory, Ubuntu Linux 2.6, Maui 3.2.6 and Torque 2.1.2. Each cluster is configured with 8 (virtual) backend nodes used by the Torque batch system. The clusters use either GT 4.0.3 or ARC 0.5.56 as Grid middleware. For both middleware configurations, one of the clusters also serve as index server for itself and the other clusters. To enable advance reservations, the WS-Agreement services are deployed on each of the four clusters.

Two sets of campus computer laboratories were used as the DiPerF testers (clients), all computers running Debian Linux 3.1. Sixteen of these computers are equipped with AMD Athlon 64 2 GHz dual core CPUs and 2 GB memory, the other sixteen have 2.8 GHz Pentium 4 CPUs with 1 GB memory each. The job submission service itself was deployed on a computer with a 2 GHz AMD Opteron CPU and 2 GB memory, running Debian Linux 3.1. All machines in the test environment are interconnected with a 100 Mbit/s network.

The job submission service was configured with a timeout of 15 seconds for all interactions with the information systems of the resources. The Grid middlewares generated updated resource information every 60 seconds and the information gathered by the broker was hence cached for this amount of time. Queries about resource information and negotiations of advance reservations were both performed using four parallel threads.

The use of a relatively small but controlled environment for tests, gives the advantage that we know that there is no background load on the clusters. Hence, the performance of the job submission service can be significantly more accurately analyzed than it could have been if evaluated in a large production Grid (e.g, as performed in [18]).

5.2. Performance results

Tests have been performed with the number of resources varying between one and four, and the number of clients being $\{3, 5, 7, 10, 15\}$. Each test starts with one client, and then another client is added every 30th second until the selected number of clients is reached. Each client executes for 15 minutes and submits trivial `/bin/true` jobs, that do not require any input or output file staging. Hence, also tests with large number of clients include time periods where smaller number of clients are used. The reason for this strategy is to better identify the relation between service load and throughput or response time.

In the following presentation, the performance results are grouped by the Grid middleware used, i.e., GT4 and ARC. For each middleware, results are presented separately for tests using the Torque "PBS" scheduler and POSIX "Fork" as execution backends.

Our results show that the performance varies very little with the number of Grid resources used. Resource discovery takes longer when more resources are used, but the load distribution of the jobs across more machines does, on the other hand, give faster response time in the dispatch step. These two factors seem to compensate each other rather well for one to four resources. Because of this, we here only present results obtained using four resources. From our tests, we also find it sufficient to present results for tests using 3, 7, and 15 clients.

For tests using the GT4 middleware, Figure 5 shows how the service throughput (lines marked with "$\times$") and response time for the job requests (lines without "$\times$") vary during the tests. The three figures present, from top to bottom, the results obtained using 3, 7, and 15 clients. Solid and dashed lines are used to represent results obtained using Fork and PBS, respectively. Notably, the left-hand scales in the figures denote response times and the right-hand ones throughputs.

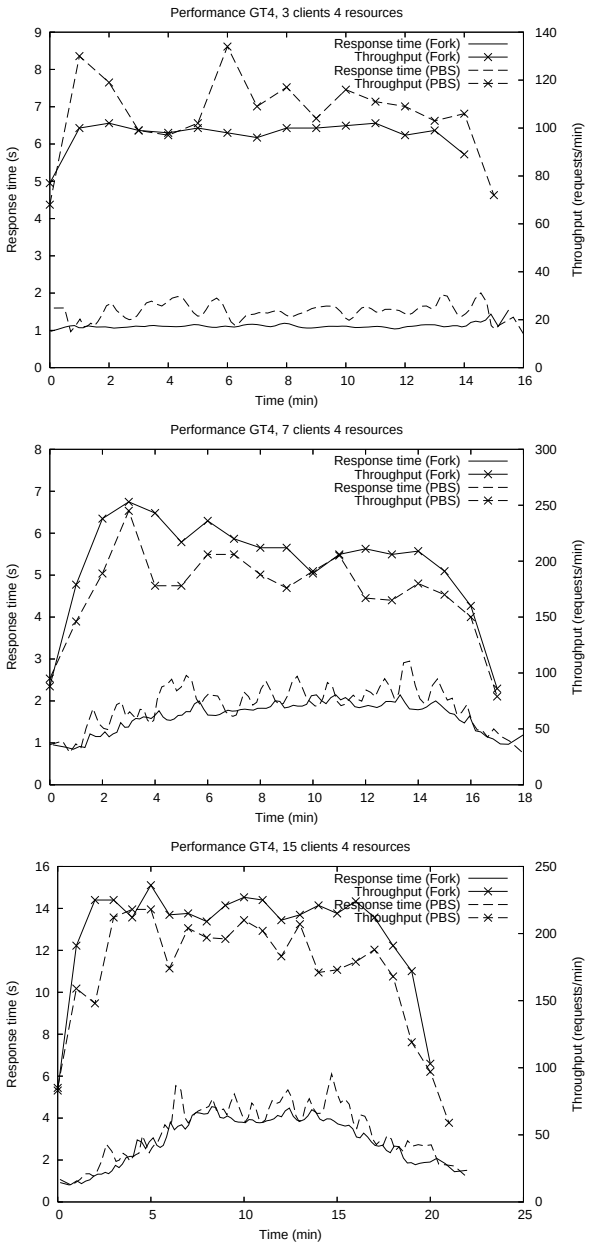


Figure 5. Performance results for GT4 using 4 resources.

In the results obtained using three clients (the topmost diagram in Figure 5), we do not see any particular trend in the results as the number of clients are increased from one to three (recall that in each test, a new client is started every 30 seconds), which indicates that the service can handle this load without problems. Notably, the response time for individual jobs is as low as down to under one second at best. As the number of clients increases to 7 in the middle graph, we observe that both the response time and the throughput increase as more clients are being started, until it reaches a maximum and then starts to decrease as the clients finish executing. The increase in response time indicate that some bottleneck has been found. As the throughput still increases, our interpretation is that the increase in response time is due to increased waiting for resources to respond, and not due to too high load for the job submission service itself.

We remark that this is the test for which we see the highest throughput for GT4, with a maximum of just over 250 jobs per minute for Fork and only slightly lower with PBS. Response times for Fork vary between one and two seconds, whereas they fluctuate up to three seconds for PBS. In comparison to the results for three clients, we see that the throughput doubles for Fork, whereas the increase in throughput for PBS is somewhat lower. When further increasing the load to 15 clients, we see that the throughput from the tests with 7 clients is maintained also for heavy load, even though we do not reach the same peak result. In summary, the tests with GT4 resources show that the job submission service is capable of handling throughput just over 250 jobs per minute and to achieve individual job response times down to under one second.

For tests using the ARC middleware, Figure 6 shows the performance using four resources and 3, 7, and 15 clients, respectively. Here, the throughput increases from 60-70 jobs/minute with three clients (the top diagram in Figure 6) to approximately 170 jobs per minute with 7 clients (the middle plot in Figure 6), while keeping response times between two and three seconds per submitted job. When further increasing the load to 15 clients, we see in the bottom diagram in Figure 6 a slight increase in throughput, to approximately 200 jobs per minute, whereas the response time increases as well, to approximately four seconds. This suggests that the maximum throughput is around 200 jobs per minute when using ARC.

Notably, in our tests PBS and Fork perform reasonably equal for both middlewares and for all combinations of different numbers of clients and resources, even though we see slightly more fluctuating response times using PBS than with Fork. However, if tests are done with jobs that require substantial computational capacity, the performance obtained using Fork will substantially decrease. For PBS, we expect the results to be similar also for more demanding jobs, if the clusters make use of real (and not virtual) back-end nodes.

The slightly more fluctuating response times obtained with PBS can also be explained by the fact that the information systems used by ARC and GT4 both perform extensive parsing of PBS log files to determine the current load on the resource. During significant load, this may occasionally lead to slow response times for resource information queries. This does in turn result in slower response times for jobs for which the broker can not use cached resource information.

During the period of constant load (while all clients execute), we see a slight decrease in throughput over time for both ARC and GT4. This decrease is most clearly visible in the tests using 7 and 15 clients. We initially suspected that this performance decline was due to scalability issues in the GT4 delegation service, investigated in [29]. During our performance tests of the delegation service (with a test setup similar to the job submission service tests) we observed a performance decline during heavy load. However, as the throughput of the delegation service is around three times higher than that of the

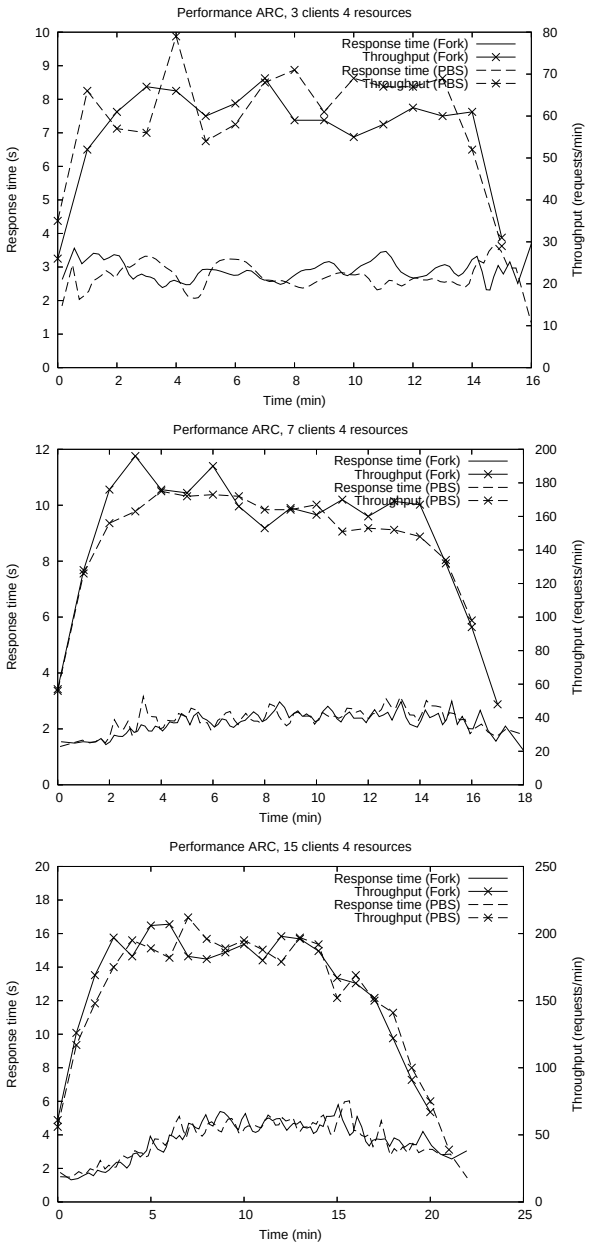


Figure 6. Performance results for ARC using 4 resources.

job submission service for similar loads, this performance decline is negligible. We also noted that the delegation service response time typically is between 0.3 and 0.8 seconds. This is a substantial part of the job submission time, especially considering that the job submission service can submit a job in less than one second, credential delegation included.

5.2.1. Advance reservations

In order to evaluate the impact of advance reservations on the job submission service performance, tests with jobs requesting reservations are compared to the corresponding tests performed without use of reservations. The performance of the job submission service for jobs using reservations are, of course, expected to be lower. A job submitted with an advance reservation requires two additional round trips (get agreement template, create agreement) during brokering and one more round trip during job dispatch (confirm temporary reservation). When each job submission request takes longer to serve, fewer jobs can utilize cached resource information before the cache expires, which further decreases performance.

As previous research has demonstrated [22, 66], the usage of advance reservations imposes a performance penalty, and does typically reduce batch system utilization dramatically already when only 20 percent of all jobs use advance reservations. Our resource brokering algorithms described in Section 3.1, are able to create reservations for all resources of interest (or a subset thereof), and upon job submission release all reservations but the one for the selected resource. However, as long as batch systems do not provide a lightweight reservation mechanism, we argue that this feature should be used only when needed.

In order to investigate the performance impact of the advance reservation mechanism, we consider a scenario where exactly one reservation is created for each submitted job. The performance results for GT4 with reservations (dashed lines) is compared to corresponding results without reservations (solid lines) in Figure 7. We note that the throughput (marked with "$\times$") with reservations is about 40 submitted jobs per minute for all three tests. In these tests, the response time increases from about five seconds (3 clients), to ten seconds (7 clients), and finally to around 20 seconds (15 clients). In comparison, for jobs submitted without reservations, the throughput increases from around 100 jobs per minute (3 clients), to around 210 jobs per minute (7 clients), and finally increases a bit more to around 220 jobs per minute when 15 clients are used. The response times for these jobs are around two seconds (both 3 and 7 clients) and three seconds for 15 clients.

The performance results for tests of jobs with advance reservations submitted to ARC are very similar to the corresponding tests with GT4, and graphs for these tests are hence omitted. For jobs with reservations submitted to ARC, the throughput is around 30 jobs per minute for 3 clients, and 40 jobs per minute for 7 and 15 clients. The response time varies from around six seconds for 3 clients, to ten seconds for 7 clients and 20 seconds for 15 clients. In the reference tests where no jobs used reservations, the throughput is around 55 jobs per minute for 3 clients, 140 jobs per minute with 7 clients and 160 jobs per minute with 15 clients. In these tests, the response time is around three seconds for both 3 and 7 clients, and around four seconds for 15 clients.

From the results for GT4 and ARC, we conclude that for jobs submitted with advance reservations, the job submission service and the WSAG services can serve around 40 submitted jobs per minute and that the average response time for these jobs is (at best) below five seconds.

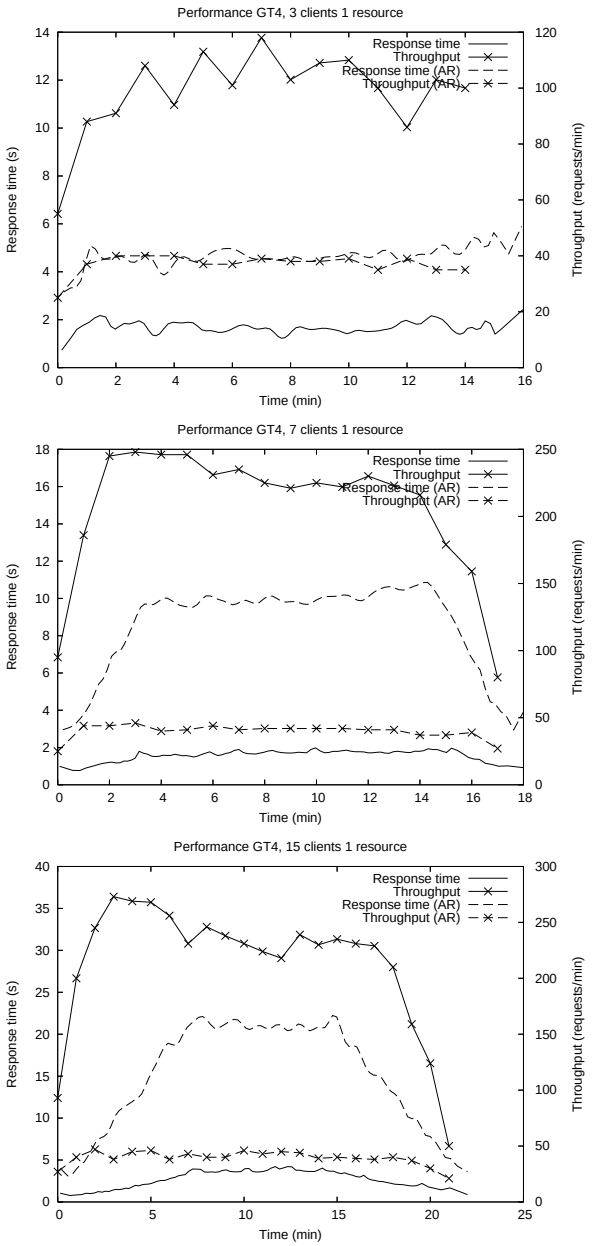


Figure 7. Performance results for advance reservations using GT4 and 1 resource.

5.2.2. *Coallocation*

The most time consuming parts of Algorithm 1 are creation of new reservations and modifications of existing ones, i.e., steps 9 and 7. The update procedure for augmenting paths (Algorithm 2) performs a series of reservation modifications and it follows from the design that its execution time increases linearly with the length of the augmenting path. The overall performance of the coallocation algorithm thus depends on how often these three mechanisms (create new reservation, modify existing reservation, path augmentation) are used. These numbers increase with the number of iterations of the algorithm that are executed. The number of iterations in turn depends on multiple factors, including the number of requested jobs, how many resources that are capable of executing each job, the degree of overlap in these sets of resources, the current load of each resource (most notably, the fragmentation of the backfill windows of the local schedulers), etc.

We have performed a series of tests to demonstrate the robustness of the coallocation algorithm, and illustrate how it performs for a given combination of coallocation request type and Grid infrastructure configuration. The results from these tests increase the understanding of the characteristics of the algorithm, but cannot, due to the intrinsic performance dependencies discussed above, be generalized beyond the particular configuration used in the tests.

The coallocation tests are performed in the test environment described in Section 5.1. In each test, background loads are created on all machines, a coallocation request is issued, and results from the coallocation algorithm execution are gathered. In all tests, four jobs are requested to be coallocated over four machines. The length of the job start time window is four hours, the maximum allowed job start time deviation (ϵ) is five minutes and the requested length of each job is one hour. Each machine has a 20 minutes long reservation as background load that is randomly placed within the four hour job start time window. Notably, this setup means that it is not always possible, even in theory, to fulfill the coallocation request.

As the machines in the test environment are identical, the path augmentation procedure is per default never required for successful coallocation. For this reason, the tests are divided into two sets: tests where path augmentation is not required, and tests where heterogeneity in the test environment is simulated to necessitate path augmentation. We refer to these as *homogeneous tests* and *heterogeneous tests*, respectively. In the homogeneous tests, each of the four requested jobs can make use of all four resources. The same hold for three of the jobs in the heterogeneous tests, whereas the fourth job in these tests can execute on only one of the machines.

5.2.2.1. Homogeneous tests. A summary of 1000 homogeneous tests, divided into successful and failed coallocation requests, is shown in Table I. This table shows the average, minimum and maximum value for the following metrics: execution time of the coallocation algorithm, the number of iterations performed, and the usage frequency of the three most time consuming operations (create new reservation, update existing reservation, and path augmentation). The table is divided into results for successful invocations of the coallocation algorithm and results for cases where the algorithm does not manage to obtain a coallocation.

Table I shows that the successful coallocation attempts are faster to execute than the failed ones. This is due to the fact that the algorithm stops when a coallocation is found, whereas the failed attempts have to scan the complete job start time window before concluding that no coallocation is possible. From the average values of the execution time and the operation counts, we conclude that the coallocation

Table I. Results for 1000 homogeneous tests of the coallocation algorithm. The term AR denotes advance reservation.

Successful coallocations (86.0%)			
	average	min	max
execution time (s)	15.02	3.30	38.65
# iterations	3.02	1	7
# new ARs	4.79	4	10
# failed new ARs	19.89	0	59
# modified ARs	4.09	0	12
# failed modified ARs	0.79	0	6
# path augmentations	0	0	0
# failed path augmentations	2.01	0	6
Failed coallocations (14.0%)			
	average	min	max
execution time (s)	27.91	13.77	39.92
# iterations	5.04	4	7
# new ARs	6.64	2	10
# failed new ARs	40.64	24	59
# modified ARs	6.71	3	11
# failed modified ARs	3.75	0	7
# path augmentations	0	0	0
# failed path augmentations	5.02	1	7

algorithm is able to perform slightly more than two advance reservation operations (create or modify) per second. This observation takes into account that a modify request performs two service invocations (removal of an old reservation and creation of a new one, see Section 3.3.2), and holds both for the failed and the successful coallocation attempts.

The successful attempts range from trivial solutions where all jobs are reserved in the first iteration and in less than four seconds, to complex scenarios where up to seven iterations with more than 60 advance reservation operations performed during 40 seconds are required to coallocate the job. The failed attempts search through the whole coallocation window, and hence show much less deviations in performance. The observable deviations are due to differences in the distribution of the background load.

5.2.2.2. Heterogeneous tests. A summary of 1000 heterogeneous tests is shown in Table II. This table has both the same division into successful and failed coallocation attempts and the same metrics as Table I. We note that also for the heterogeneous tests, the successful attempts are faster and the coallocation algorithm carries out just above two advance reservation operations per second. It follows from the construction of the coallocation algorithm that, as only one job in the heterogeneous tests may require path augmentation, the maximum number of successful path augmentations is one. For the successful attempts, the average number of successful path augmentations is 0.81. The reason for this

Table II. Results for 1000 heterogeneous tests of the coallocation algorithm. The term AR denotes advance reservation.

Successful coallocations (66.3%)			
	average	min	max
execution time (s)	9.21	3.32	22.56
# iterations	1.97	1	4
# new ARs	3.65	3	7
# failed new ARs	7.44	0	27
# modified ARs	1.82	0	7
# failed modified ARs	0.46	0	4
# path augmentations	0.81	0	1
# failed path augmentations	0.67	0	3
Failed coallocations (33.7%)			
	average	min	max
execution time (s)	15.71	7.62	23.40
# iterations	3.30	2	4
# new ARs	5.42	1	8
# failed new ARs	16.32	3	32
# modified ARs	3.28	0	7
# failed modified ARs	2.77	0	5
# path augmentations	0	0	0
# failed path augmentations	2.59	0	4

number being less than one is that in some tests, none of the first three jobs reserve the only resource that the forth job can use, and path augmentation is hence not required. Failed path augmentations occur, when an augmenting path is found and augmentation hence appears possible (Step 15 of Algorithm 1), but a background load job prevents the new reservation (Step 1 of Algorithm 2) from being created. We note that all failed coallocation attempts have in common that no path augmentation operation is successful.

Although results for the homogeneous and the heterogeneous tests are not directly comparable with each other due to the intrinsic performance dependencies of the coallocation algorithm, we observe that, as the number of potential matchings between jobs and resources are fewer in the heterogeneous tests, these tests are faster to execute than the homogeneous ones.

Tables I and II list the execution time of the coallocation procedure as described in Algorithm 1. In a complete job coallocation scenario, tasks such as job submission service invocation, job request validation, resource discovery and information retrieval must also be performed. These tasks are similar to the initial steps executed during submission of individual jobs and, as discussed earlier, take approximately one to three seconds to execute if new Grid resource information must be retrieved and a less than a tenth of that time if cached resource information is available.

In addition to the above performance observations, the coallocation evaluation also demonstrates that the implementation of the coallocation algorithm is robust, as no errors (except for failed coallocation

attempts, that are not errors per se) occurred during the 2000 tests, that had a total execution time of more than seven hours.

6. RELATED WORK

We have identified a number of contributions related to our work on Grid resource brokering, including performance prediction for Grid jobs, the usage of advance reservations and coallocation in Grids, and Grid interoperability efforts. In the following, we make a brief review of these.

6.1. General resource brokering

The composable ICENI Grid scheduling architecture is presented in [77], together with a performance comparison between four Grid scheduling algorithms; random, simulated annealing, best of n random, and a game theoretic approach. The eNANOS Grid resource broker [60] supports submission and monitoring of Grid jobs. Features include usage of the GLUE information model [4] and a mechanism where users can control the resource selection by weighting the importance of attributes such as CPU frequency and RAM size.

There are a number of projects that investigate market-based resource brokering approaches. These approaches may typically have a starting point in bartering agreements, in pre-allocations of artificial *Grid credits* or be based on real economical compensation. In such a Grid marketplace, resources can be sold either at fixed or dynamic prices, e.g., in a strive for a supply and demand equilibrium [75]. Claimed advantages of the economic scheduling paradigm include load balancing and increased resource utilization, both a result of good balance between supply and demand for resources [75]. Examples of work on economic brokering include [10, 12, 20, 52]. An alternative to market-based economies is Grid-wide fairshare scheduling [16] that can be viewed as a planned economy.

6.2. Performance prediction

One method for selecting the submission target for a computational job is to predict the performance of the job on each resource of interest. These predictions can include the job start time as well as the job execution time. Techniques for such predictions include (i) applying statistical models to previous executions [2, 38, 43, 65, 70] and (ii), heuristics based on job and resource characteristics [34, 44, 74].

In our previous work [19], we use a hybrid approach. The performance characteristics of an application is classified using computer benchmarks relevant for the application, as in method (ii). When predicting the performance for a Grid resource, the benchmark results for this machine is compared with those of a reference machine where the application has executed previously. This comparison with earlier execution of the application reuses techniques from method (i).

A Grid application performance model similar to TTD is described by Ali et al. [1]. In this work, application execution time and batch queue waiting time are both predicted using method (i), whereas file transfer times are estimated from file size and bandwidth information.

6.3. Interoperability efforts

There are several resource brokering projects which target resources running different Grid middlewares, e.g., Gridbus [72], which can schedule jobs on resources running, e.g., Globus [30], UNICORE [68], and Condor [45]. The GridWay project [36] targets resources running both protocol oriented (GT2) and service-based versions (GT4) of the Globus toolkit as well as LCG [40]. One difference between our contribution and these projects is that we target the use of any Grid middleware both on the resource and client side by allowing clients to express their jobs in the native job description language of their middleware (or directly in JSDL), whereas the respective job description languages of Gridbus and GridWay are fixed on the client side.

In the contribution by Pierantoni et al. [57], Metagrid Services are used as a bridge between users and different Grids. In this architecture, condensed graphs are used to express workflows of jobs. Interoperability is demonstrated among WebCom (a workflow engine), GT4, and LCG2 [57]. Similar ideas are explored by Kertész et al. [39], who define an architecture for a meta-broker and a language for communicating broker requirements in addition to job requirements. Instead of performing resource selection, such a meta-broker selects the best Grid resource broker and hence creates a hierarchy of Grid brokers. Common features in our contribution and the work by Kertész et al. is the use of JSDL and a plugin-based architecture for interaction with specific middlewares. The UniGrids project [71] specially targets interoperability between the Globus [30] and UNICORE [68] middlewares. The Grid Interoperability Now (GIN) [31] initiative focuses on establishing islands of interoperation between existing Grid resources, and growing those islands to achieve an increasing set of interoperable Grids. The goal of the Open Middleware Infrastructure Institute (OMII) Europe [54] is to make components for job management, e.g., the OGSA Basic Execution Service [33]; data integration; and accounting available for multiple platforms, including gLite [14], Globus [24], and UNICORE [68].

There are some projects that have adopted JSDL to describe jobs, e.g., [32, 39, 50].

6.4. Advance reservations

Several contributions conclude that an advance reservation feature is required to meet QoS guarantees in Grid environments [26, 35, 63]. Unfortunately, the support for reservations in the underlying infrastructure is currently limited. Qu describes a method to overcome this shortcoming by adding a Grid advance reservation manager on top of the local scheduler(s) [58]. Advance reservations can hence be provided regardless of whether the local scheduler supports them. This reservation approach however requires that all job requests are passed through the Grid advance reservation manager.

The performance penalty imposed by the usage of advance reservations (typically decreased resource utilization) has been studied [66, 67]. The work in [22] investigates how performance improvements can be can be achieved by allowing laxity (flexibility) in advance reservation start times.

Standardization attempts include [61], which defines a protocol for management of advance reservations. The more recent WS-Agreement [5] standard proposal defines a general architecture that enables two parties, the agreement provider and the agreement initiator, to enter an agreement. Although not specifically targeting advance reservations, WS-Agreement can be used to implement these, as demonstrated e.g. by [18, 48, 73].

6.5. Coallocation

The work by Czajkowski et.al. [11] describes a library for initiating and controlling coallocation requests and an application library for synchronization. By compiling an application that requires coallocation with the application library, the subjob instances can wait for each other at a barrier prior to commencing execution. This is typically required when setting up an MPI environment distributed across several machines. The work in [11] does not contain any algorithm for the actual selection of which resources to coallocate.

The Globus Architecture for Reservation and Allocation (GARA) [26] provides a programming interface to simplify the construction of application-level coallocators. GARA can perform both immediate reservations (allocations) and advance reservations. The system furthermore supports several resource types, including networks, computers and storage. The GARA project is focused on the development of a library for coallocation agents and only outlines one possible coallocation agent [26], targeting the allocation of two computer systems and an interconnection network at a fixed time. The focus of our work is the implementation of a more general coallocation service able to allocate an arbitrary number of resources. Our coallocation algorithm also differs from GARA as our algorithm allows for a flexible reservation start within a given interval of time.

The authors of the KOALA system [51] propose a mechanism for implementing coallocation without using advance reservations. Their approach is to request longer execution times than required by the jobs, and delay the start of the each job until all jobs are ready to start executing.

The work by Mateescu [46] defines an architecture for coallocation based on GT2. Mateescu's coallocation algorithm shares some concepts with our algorithm, including the use of a window of acceptable job start times and iterations in which reservations for all job requests are created. One difference is that the algorithm by Mateescu only attempts to reserve resources at a few predefined positions in the start time window, whereas our algorithm uses information included in rejection messages to dynamically determine where in the start time window to retry to create reservations. Our algorithm also tries to modify existing reservations when considering a new start time window and uses a mechanism to exchange reservations between jobs in the coallocated job, which can resolve conflicts if more than one job requests the same resource(s).

The coallocation algorithm developed by Wäldrich et al. [73] models reservations using the WS-Agreement framework and uses the concept of coallocation iterations. In each iteration of the algorithm by Wäldrich et al., a list of free time slots is requested from each local scheduler. Then, an off-line matching of the time slots with the coallocation request is performed. If the request can be mapped onto some set of resources, reservations are requested for the selected slots.

Our coallocation algorithm has some fundamental differences from the one described by Wäldrich et al. Our algorithm selects which resources to coallocate incrementally by matching one resource at the time (on-line), whereas the algorithm by Wäldrich et al. is based on an off-line calculation of which resources to use. Furthermore, our coallocation algorithm allows a user-specified fluctuation in reservation start times, while the algorithm described in [73] uses a fixed notion of reservation start time.

The advantages of on-line coallocation include fewer requirements on the local scheduler, as on-line algorithms need not know all available time slots in advance. Not all local schedulers allow users to see the current backfill-window. Furthermore, in order to be accessible by a Grid coallocation service, the backfill-window must be included in the information advertised by the Grid information system.

Information retrieved from such a system can be both incomplete and outdated. Even if the backfill-window is available and up-to-date there are possible complications. The existence of a free slot in the backfill window is not a guarantee that a certain user may reserve this slot. Any reservation request may, e.g., due to policy reasons be denied. Furthermore, nodes in a cluster can be heterogeneous in the number of cores, available memory etc. This implies that information about the backfill window is not enough to (off-line) match a job with specific requirements to some time slot. These complicating factors suggest, as argued in Section 2.3, that reservations should be created on a trial-and-error basis. The main benefit of off-line coallocation algorithms is that they use fewer reservations and can hence be seen as more efficient.

The off-line coallocation algorithms use methods resembling optimistic concurrency control for transactions [42], whereas on-line algorithms can be described as a more pessimistic locking approach. In the coallocation context, a lock is equivalent to a reservation. Which of the two approaches that is better from a transaction perspective much depends on the likelihood of (resource reservation) failure. Further work in the coallocation area should investigate the likelihood of failures and also leverage the theory developed for distributed transactions.

The GridARS project defines a protocol for advance reservations and coallocation of computational and network resources [69]. The protocol specifies a two-phase commit and does hence provide safe transactions for coallocation. There is however no description of the actual algorithm used to select and coreserve the resources.

The work described in [3] reuses the concept of barriers from [11]. In [3], the coallocator architecture consists of a selection agent, a request agent, and a barrier agent. A model for multistage coallocation is developed, where one coallocation service passes a subset of the coallocation request to another coallocation service, thus forming a hierarchy of coallocators. The barrier functionality developed in [3] also supports the synchronization of hierarchically coallocated jobs. Our work differs from [3], e.g., by using a flat model where a broker negotiates directly with the resources.

Deadlocks and deadlock prevention techniques in a coallocation context are described by Park et al. [56] whereas other work [9] suggests performance improvements for these deadlock prevention techniques. We, however, argue that the coallocation algorithm described in this paper does not cause deadlocks. Deadlocks can only occur when the following four conditions hold simultaneously: (i) mutual exclusion, (ii), hold and wait, (iii) no preemption, and (iv), circular wait [37]. Our algorithm modifies (or releases) reservations for resources whenever it fails to acquire an additional required resource. Condition (ii) does hence not hold and no deadlock can occur.

7. CONCLUSIONS

We have demonstrated how a general Grid job submission service can be designed to enable all-to-all cross-middleware job submission by leveraging emerging Grid and Web services standards and technology. The architecture's ability to manage different middlewares have been demonstrated by providing plugins for GT4 and NorduGrid/ARC. Hence, job and resource requests can be specified in any of these two input formats, and independently, the jobs can be submitted to resources running any of these middlewares.

A modular design facilitates the customizability of the architecture, e.g., for tuning the resource selection process to a particular set of Grid resources or for a specific resource brokering scenario. The

current implementation includes resource selection algorithms that can make use of, but do not depend on, rather sophisticated features for predicting individual job performance on individual resources. It also provides support for advance resource reservations and coallocation of multiple resources.

Even though the job submission service is designed for decentralized use, i.e., typically to be used by a single user or a small group of users, the performance analysis demonstrates that it can handle a quite significant load. In fact, the job submission service itself appears not to be the bottleneck as times waiting for resources becomes dominating during high load. At best, the job submission service is able to give individual job response times below one second and to provide a total throughput of over 250 jobs per minute.

The scientific contributions in this work are mainly in two directions. We conclude that the current set of Grid and Web service standards enables interoperability between different Grid middlewares, although only at a fundamental level. We have however demonstrated that cross-middleware interoperability need not be restricted to the least common denominator of the used middlewares. Even though middleware specific job description attributes are lost in translation, other mechanisms, e.g., the job preferences document used in the job submission service, allow users to express QoS requirements for their jobs. Use of proper infrastructure extensions, e.g., the WS-Agreement services, enable such requirements to be fulfilled across different middlewares.

The other direction of contributions of this work is the proposed coallocation algorithm that allows users to perform arbitrarily coordinated allocations of multiple resources. Even though currently only implemented for computational resources, the algorithm is general enough to be used to coordinate use of any reservable resource, e.g., network bandwidth. The discussion of the differences between the on-line and off-line coallocation approaches adds to the understanding of the various problems that must be addressed in a coallocation scenario.

Future directions for this work include adaptation of the current architecture and interfaces to adhere to more recent emerging standards such as the OGSA Basic Execution Service [33] and the OGSA Execution Management Services [27]. An ongoing effort is the integration of the job submission service with the LCG2/gLite middleware. As the job submission service replaces the LCG2/gLite Resource Broker component in this scenario, the only involved components are the GT2 GRAM computing element and the information system, the latter based on the Berkeley Database Information Index (BDII). The translation of the Condor-style *classads* used as job description in LCG2/gLite is rather tedious as classads do not define a schema of valid attributes but rather allow any value-pair expression.

We also plan to develop a library for job coordination of coallocated jobs, allowing the jobs to coordinate themselves prior to execution at their respective cluster. This is required, e.g., for setting up MPI environments for jobs using cross-cluster communication. This work will build on our experiences from job coallocation and previous work, such as [11]. Based on our earlier experiences with Grid workflows [17], we currently investigate how the coallocation algorithm can be used to improve QoS for job pipelines in a Grid data-flow scenario.

8. SOFTWARE AVAILABILITY

The software described in this paper is available at `www.gird.se/jss`. This web page contains the job submission service software, installation instructions and a user's guide.

ACKNOWLEDGEMENTS

The authors are grateful to Catalin Dumitrescu, Peter Gardfjäll, Klas Markström, Ioan Raicu, Åke Sandgren, and Björn Torkelsson. We also acknowledge the three anonymous referees for constructive comments. This research was conducted using the resources of the High Performance Computing Center North (HPC2N).

REFERENCES

1. A. Ali, A. Anjum, T. Azim, J. Bunn, A. Mehmood, R. McClatchey, H. Newman, W. ur Rehman, C. Steenberg, M. Thomas, F. van Lingen, I. Willers, and M.A. Zafar. Resource management services for a Grid analysis environment. In W-C. Chun and J. Duato, editors, *Proceedings of the 2005 International Conference on Parallel Processing*, pages 53–60. IEEE Computer Society, 2005.
2. 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*, 2004.
3. S. Ananad, S. Yoginath, G. von Laszewski, and B. Alunkal. Flow-based multistage co-allocation service. In B.J. d'Auriol, editor, *Proceedings of the International Conference on Communications in Computing*, pages 24–30, Las Vegas, 2003. CSREA Press.
4. S. Andreozzi, S. Burke, L. Field, S. Fisher, B. Kónya, M. Mambelli, J. M. Schopf, M. Viljoen, and A. Wilson. Glue schema specification version 1.3. http://glueschema.forge.cnaf.infn.it/Spec/V13, January 2009.
5. A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web services agreement specification (WS-Agreement). https://forge.gridforum.org/sf/docman/do/downloadDocument/projects.graap-wg/docman.root.current_drafts/doc6090, November 2006.
6. A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web Services Agreement Negotiation Specification (WS-AgreementNegotiation). https://forge.gridforum.org/sf/go/doc6092?nav=1, November 2006.
7. 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, November 2006.
8. Apache Web Services Project - Axis. http://ws.apache.org/axis, January 2009.
9. D. Azougagh, J-L. Yu, J-S. Kim, and S-R. Maeng. Resource co-allocation: a complementary technique that enhances performance in Grid computing environment. In L. Barolli, editor, *Proceedings of the eleventh International Conference on Parallel and Distributed Systems*, pages 36–42, 2005.
10. R. Buyya, D. Abramson, J. Giddy, and H. Stockinger. Economic models for resource management and scheduling in Grid computing. *Concurrency Computat.: Pract. Exper.*, 14:1507–1542, 2002.
11. K. Czajkowski, I. Foster, and C. Kesselman. Resource co-allocation in computational Grids. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC-8)*, pages 219–228, 1999.
12. M. Dalheimer, F-J. Pfreundt, and P. Merz. Agent-based Grid scheduling with Calana. In R. Wyrzykowski, J. Dongarra, N. Meyer, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics, LNCS 3911*, pages 741–750. Springer Verlag, 2005.
13. C. Dumitrescu, I. Raicu, M. Ripeanu, and I. Foster. Diperf: An automated distributed performance testing framework. In *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, pages 289–296. IEEE Computer Society, 2004.
14. EGEE. gLite - lightweight middleware for grid computing. www.cern.ch/glite, August 2008.
15. 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.
16. 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, *First International Conference on e-Science and Grid Computing*, pages 221–229. IEEE CS Press, 2005.
17. E. Elmroth, F. Hernández, and J. Tordsson. A light-weight Grid workflow execution engine enabling client and middleware independence. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics, LNCS 4967*, pages 754–761. Springer Verlag, 2007.

18. 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, *First International Conference on e-Science and Grid Computing*, pages 212–220. IEEE CS Press, 2005.
19. 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*, 24(6):585–593, 2008.
20. C. Ernemann, V. Hamscher, and R. Yahyapour. Economic scheduling in Grid computing. In D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing, LNCS 2537*, pages 128–152, 2002.
21. Thomas Fahringer. *Automatic Performance Prediction of Parallel Programs*. Kluwer Academic Publishers, 1996.
22. U. Farooq, S. Majumdar, and E. W. Parsons. Impact of laxity on scheduling with advance reservations in Grids. In *MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 319–324. IEEE Computer Society, 2005.
23. International Organization for Standardization. ISO/IEC 2382-1 information technology - vocabulary - part 1: Fundamental terms, 1993.
24. 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.
25. 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. www.ibm.com/developerworks/library/ws-resource/ws-modelingresources.pdf, January 2009.
26. I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance reservations and co-allocation. In M. Zitterbart and G. Carle, editors, *7th International Workshop on Quality of Service*, pages 27–36. IEEE, 1999.
27. 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, January 2009.
28. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
29. 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.
30. Globus. http://www.globus.org. January 2009.
31. Grid Interoperability Now. http://wiki.nesc.ac.uk/read/gin-jobs. January 2009.
32. GridSAM. http://gridsam.sourceforge.net. January 2009.
33. A. Grimshaw, S. Newhouse, D. Pulsipher, and M. Morgan. OGSA Basic Execution Service version 1.0. https://forge.gridforum.org/sf/go/doc13793, January 2009.
34. R. Gruber, V. Keller, P. Kuonen, M-C. Sawley, B. Schaeli, A. Tolou, M. Torruella, and T-M. Tran. Towards an intelligent Grid scheduling system. In R. Wyrzykowski, J. Dongarra, N. Meyer, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics, LNCS 3911*, pages 751–757. Springer Verlag, 2005.
35. M. H. Haji, I. Gourlay, K. Djemame, and P. M. Dew. A snap-based community resource broker using a three-phase commit protocol: A performance study. *The Computer Journal*, 48(3):333–346, 2005.
36. E. Huedo, R.S. Montero, and I.M. Llorente. A framework for adaptive execution on Grids. *Softw. Pract. Exper.*, 34(7):631–651, 2004.
37. E. G. Coffman Jr, M. J. Elphick, and A. Shoshani. System deadlocks. *Computing Surveys*, 3(2):68–78, 1971.
38. 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.
39. A. Kertész 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'07 conference, Dresden, Germany*, 2007.
40. J. Knobloch and L. Robertson. LHC computing Grid technical design report. http://lcg.web.cern.ch/LCG/tdr/, January 2009.
41. K. Krauter, R. Buyya, and M. Maheswaran. A taxonomy and survey of Grid resource management systems for distributed computing. *Softw. Pract. Exper.*, 15(32):135–164, 2002.
42. H.T. Kung and J.T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.
43. H. Li, J. Chen, Y. Tao, D. Gro, and L. Wolters. Improving a local learning technique for queue wait time predictions. In S.J. Turner, B.S. Lee, and W. Cai, editors, *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2006)*, pages 335–342. IEEE Computer Society, 2006.

44. H. Li, D. Groep, and L. Wolters. Efficient response time predictions by exploiting application and resource state similarities. In *6th International Workshop on Grid Computing (GRID 2005)*, pages 234–241, 2005.
45. M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, June 1988.
46. 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, 2003.
47. Maui Cluster Scheduler. http://www.clusterresources.com/products/maui/, January 2009.
48. A. S. McGough, A. Afzal, J. Darlington, N. Furmento, A. Mayer, and L. Young. Making the Grid predictable through reservations and performance modelling. *The Computer Journal*, 48(3):358–368, 2005.
49. D. Mills. Network time protocol (version 3) specification, implementation and analysis. http://www.ietf.org/rfc/rfc1305.txt, January 2009.
50. K. Miura. Overview of japanese science Grid project NAREGI. *Progress in Informatics*, 1(3):67–75, 2006.
51. H. H. Mohamed and D. H. J. Epema. Experiences with the KOALA co-allocating scheduler in multiclusters. In *Proceedings of the International Symposium on Cluster Computing and the Grid (CCGRID2005)*, pages 784–791. IEEE Computer Society, 2005.
52. R. Moreno and A. B. Alonso-Conde. Job scheduling and resource management techniques in economic Grid environments. In F. Fernández Rivera, M. Bubak, A. Gómez Tato, and R. Doallo, editors, *Grid Computing - First European Across Grids Conference, LNCS 2970*, pages 25–32, 2004.
53. F. Nadeem, M. M. Yousaf, R. Prodan, and T. Fahringer. Soft benchmarks-based application performance prediction using a minimum training set. In *In Second International Conference on e-Science and Grid computing, Amsterdam, Netherlands*. IEEE press, 2006.
54. OMII Europe. http://omii-europe.org, October 2008.
55. Open Grid Forum. www.ogf.org. January 2009.
56. J. Park. A scalable protocol for deadlock and livelock free co-allocation of resources in internet computing. In S. Helal, Y. Oie, C. Chang, and J. Murai, editors, *Symposium on Applications and the Internet, SAINT 2003*, pages 66–73. IEEE Computer Society, 2003.
57. 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*, 2006.
58. C. Qu. A Grid advance reservation framework for co-allocation and co-reservation across heterogeneous local resource management systems. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics, LNCS 4967*, pages 770–779. Springer Verlag, 2007.
59. K. Ranganathan and I. Foster. Simulation studies of computation and data scheduling algorithms for data grids. *Journal of Grid computing*, 1(1):53–62, 2003.
60. 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.
61. A. Roy and V. Sander. Advance reservation API. http://www.ogf.org/documents/GFD.5.pdf, January 2009.
62. 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.
63. M. Siddiqui, A. Villazon, and T. Fahringer. Grid capacity planning with negotiation-based advance reservation for optimized QoS. In *Proceedings of the 2006 ACM/IEEE SC—06 Conference (SC'06)*, 2006.
64. 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, LNCS 1459*, pages 122–142, 1999.
65. W. Smith, I. Foster, and V. Taylor. Using run-time predictions to estimate queue wait times and improve scheduler performance. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing, LNCS 1459*, pages 202–219, 1999.
66. W. Smith, I. Foster, and V. Taylor. Scheduling with advance reservations. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium*, pages 127–132. IEEE, 2000.
67. Q. Snell, M. Clement, D. Jackson, and C. Gregory. The performance impact of advance reservation meta-scheduling. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing: IPDPS 2000 Workshop, JSSPP 2000, LNCS 1911*, pages 137–153. Springer Verlag, 2000.
68. 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.
69. A. Takefusa, H. Nakada, T. Kudoh, Y. Tanaka, and S. Sekiguchi. GridARS: An advance reservation-based Grid co-allocation framework for distributed computing and network resources. In E. Frachtenberg and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing, LNCS 4942*, pages 152–168. Springer Verlag, 2007.

70. D. Tsafrir, Y. Etsion, and D. G. Feitelson. Modeling user runtime estimates. In D.G. Feitelson and. E. Frachtenberg, L. Rudolph, and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing, LNCS 3834*, pages 1–35, 2005.
71. UniGrids. www.unigrids.org. January 2009.
72. 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, 2006.
73. O. Wäldrich, P. Wieder, and W. Ziegler. A meta-scheduling service for co-allocating arbitrary types of resources. In R. Wyrzykowski, J. Dongarra, N. Meyer, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics, LNCS 3911*, pages 782–791. Springer Verlag, 2005.
74. J. Wang, L-Y. Zhang, and Y-B. Han. Client-centric adaptive scheduling for service-oriented applications. *J. Comput. Sci. and Technol.*, 21(4):537–546, 2006.
75. R. Wolski, J. Brevik, J. S. Plank, and T. Bryan. Grid resource allocation and control using computational economies. In F. Berman, G. Fox, and A. Hey, editors, *Grid Computing: Making The Global Infrastructure a Reality*, chapter 32. John Wiley & Sons, 2003.
76. R. Yahyapour. Considerations for resource brokerage and scheduling in Grids. In G. R. Joubert, W. E. Nagel, F. J. Peters, and W. V. Walter, editors, *Parallel Computing: Software Technology, Algorithms, Architectures and Applications, PARCO 2003, Dresden, Germany*, pages 627–634, 2004.
77. L. Young, S. McGough, S. Newhouse, and J. Darlington. Scheduling architecture and algorithms within the ICENI Grid middleware. In S. Cox, editor, *Proceedings of the UK e-Science All Hands Meeting*, pages 5–12, 2003.

IV

Paper IV

Designing General, Composable, and Middleware-independent Grid Infrastructure Tools for Multi-tiered Job Management*

E. Elmroth, P. Gardfjäll, A. Norberg, J. Tordsson, and P-O. Östberg

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

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

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

* By permission of Springer-Verlag

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

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

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

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

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

1. Introduction

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

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

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

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

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

2. A Model for Multi-Tiered Job Submission Architectures

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

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

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

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

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

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

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

3. The Grid Job Management Framework (GJMF)

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

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

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

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

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

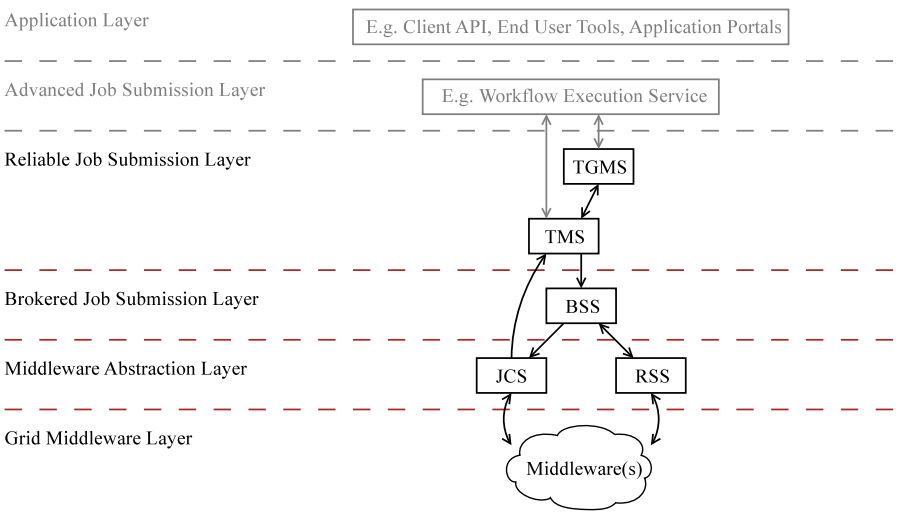


Figure 1. GJMF components mapped to their respective architectural layers.

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

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

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

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

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

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

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

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

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

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

4. Performance Evaluation

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

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

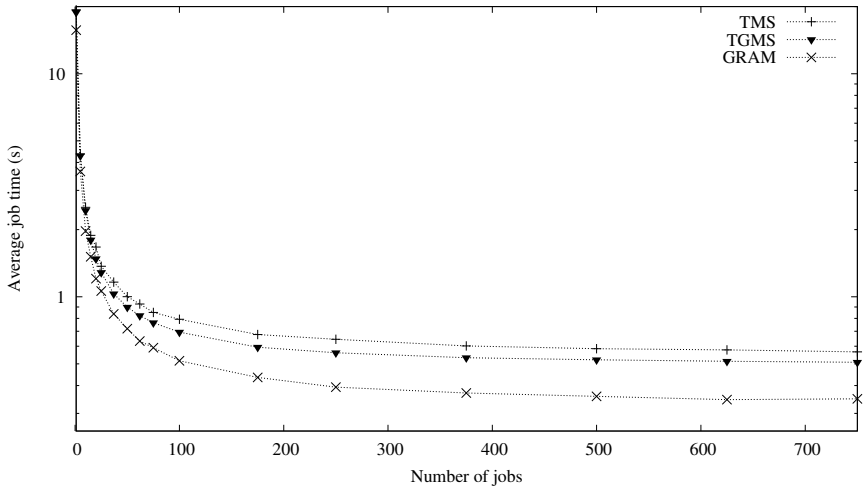


Figure 2. GRAM and GJMF job processing performance.

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

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

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

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

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

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

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

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

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

5. Related Work

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

6. Concluding Remarks

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

Acknowledgments

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

V

Paper V

A Light-weight Grid Workflow Execution Service Enabling Client and Middleware Independence*

E. Elmroth, F. Hernández, and J. Tordsson

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

Abstract: We present a generic and light-weight Grid workflow execution engine made available as a Grid service. A long-term goal is to facilitate the rapid development of application-oriented end-user workflow tools, while providing a high degree of Grid middleware-independence. The workflow engine is designed for workflow execution, independent of client tools for workflow definition. A flexible plugin-structure for middleware-integration provides a strict separation of the workflow execution and the processing of individual tasks, such as computational jobs or file transfers. The light-weight design is achieved by focusing on the generic workflow execution components and by leveraging state-of-theart Grid technology, e.g., for state management. The current prototype is implemented using the Globus Toolkit 4 (GT4) Java WS Core and has support for executing workflows produced by Karajan. It also includes plugins for task execution with GT4 as well as a high-level Grid job management framework.

* By permission of Springer-Verlag

A Light-Weight Grid Workflow Execution Engine Enabling Client and Middleware Independence*

Erik Elmroth, Francisco Hernández, and Johan Tordsson

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

Abstract. We present a generic and light-weight Grid workflow execution engine made available as a Grid service. A long-term goal is to facilitate the rapid development of application-oriented end-user workflow tools, while providing a high degree of Grid middleware-independence. The workflow engine is designed for workflow execution, independent of client tools for workflow definition. A flexible plugin-structure for middleware-integration provides a strict separation of the workflow execution and the processing of individual tasks, such as computational jobs or file transfers. The light-weight design is achieved by focusing on the generic workflow execution components and by leveraging state-of-the-art Grid technology, e.g., for state management. The current prototype is implemented using the Globus Toolkit 4 (GT4) Java WS Core and has support for executing workflows produced by Karajan. It also includes plugins for task execution with GT4 as well as a high-level Grid job management framework.

1 Introduction

Motivated by the tedious work required to develop end-user workflow tools and the lack of generic tools to facilitate such development, this contribution focus on a light-weight and Grid-interoperable workflow execution engine made available as a Grid service. As a point of departure, we identify important and generic capabilities supported by well-recognized complete workflow systems [18, 15, 9, 14, 1, 10, 2] (e.g., workflow design, workflow repositories, information management, workflow execution, workflow scheduling, fault tolerance, and data management). However, many of these projects provide similar functionality and much work is overlapping, as the systems have been developed independently [18].

The tool presented here is not proposed as an alternative to these more complete workflow systems, but as a core component for developing new end-user tools and problem solving environments. The aim is to offer a generic workflow

* 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.

execution engine that can be employed for building new high-level tools as well as to provide support for both processing individual tasks on multiple Grid middlewares and accepting different workflow languages as input. The engine is light-weight as it focuses only on workflow execution (i.e., selecting tasks that are ready to execute) and its corresponding state management.

The engine is developed with a strict focus on Grid resources for task processing and makes efficient use of state-of-the-art Web and Grid services technology. The current prototype is implemented using the Java WS Core from the Globus Toolkit 4 (GT4) [7]. The service has support for executing workflows expressed either in its native workflow language or the Karajan [16] format. It includes plugins for arbitrary Grid tasks, e.g., for execution of computational tasks in GT4 and in the high-level Grid Job Management Framework (GJMF) [3, 5], as well as GridFTP file transfers.

2 System and Design Requirements

The general system requirements follow directly from the aim and motivation for the proposed workflow engine. As it is developed with a general aim to provide an efficient and reusable tool for managing workflows in Grid environments, overall requirements include client and middleware independence, modularity, customizability, and separation of concerns [4]. A set of high-level design requirements for Grid workflow systems includes the following.

- The *workflow execution* should be separated from the *workflow definition*. The former must be done by the engine, the latter can be done, e.g., by an application specific GUI or a Web portal. Furthermore, workflow repositories and application specific information should not be managed by the service.
- The workflow engine should be independent of the Grid middleware used to execute the tasks, with middleware-specific interactions performed by plugins. The plugins should in turn be unaware of the context (the workflow) to which individual Grid jobs belong.
- The design can and should to a large extent leverage state-of-the-art Grid technology and emerging standards, e.g., by making use of general features of the Web Services Resource Framework (WSRF) [13] instead of implementing their workflow-specific counterparts.
- The engine should have a clean separation between the state management and the handling of task dependencies.

In addition to the high-level design requirements, the following specific system requirements are highlighted. The workflow system should:

- provide support for executing workflows, managing workflow state, and pausing and resuming execution. This enables restart of partially completed workflows stored on disk, and provides a foundation for fault tolerant workflow execution.

- provide support for both abstract (resources unspecified) and concrete workflows (resources specified on a per-task level) as well as arbitrary nestings of workflows.
- provide support for *dynamic workflows*, i.e., making it possible to modify an already executing workflow, by pausing the execution before modification.
- provide support for workflow monitoring, both synchronously and by asynchronous notifications.
- provide support for notifications of different granularity, e.g., enabling asynchronous status updates on both a per workflow and a per task basis.

These requirements are in agreement with and extend on the requirements of Grid workflow engines presented in [6]. How the requirements are mapped to the actual implementation is presented in Section 3.

3 Design and Implementation

The design requirements of customizability and ability for integration with different client tools and middlewares are met by use of appropriate plugin points. The chain-of-responsibility design pattern allows concurrent usage of multiple implementations of a particular plugin. The three main responsibilities of the workflow service, namely management of task dependencies (i.e., deciding the task execution order), execution of workflow tasks on Grid resources, and management of workflow state, are each performed by separate modules.

Reuse, in a broad sense, is a key issue in the design. The workflow service reuses ideas from an architecture for interoperable Grid components [5] and builds on a framework for managing stateful Web services and notifications [13]. Exploiting the capabilities offered by GT4 Java WS Core (e.g., security and persistency) also simplifies the design and implementation of the service.

3.1 Modelling Workflows with the WSRF

The workflow engine uses the tools provided by GT4 Java WS Core to make the engine available as a Grid service and to manage the workflow state. Building the engine on top of Java WS Core should not be interpreted as built primarily for GT4-based Grids. Integration with different middlewares is provided by middleware-specific plugins which are independent from the workflow execution.

By careful design, the service can handle arbitrarily many workflows concurrently without these interfering with each other. Multiple users can share the same workflow service, but only the creator of a workflow instance can monitor and control that workflow. Each workflow is modelled as a WS-Resource and all information about a workflow, including task descriptions, inter-task dependencies and workflow state, is stored as WS-ResourceProperties. The default behavior is to store each WS-Resource in a separate file, although alternative implementations such as persistency via database can be added easily. Reuse of

the Java WS Core persistency mechanisms makes workflow state handling trivial. Workflow state management enables the control of long-running workflows and the recovery of workflows, e.g., upon service failures.

The states handled include `default`, `ready`, `running`, and `completed`, which apply to both tasks and (sub)workflows. Tasks can also be `failed` whereas workflows can be `disabled`. All newly created tasks and workflows have the default state. A task/workflow is ready to be started when all tasks on which it depends are completed. Running tasks are processed by some Grid resource until they become either completed or failed. A running workflow has at least one task that is not completed and no failed task, whereas completed workflows only contain completed tasks/subworkflows. A workflow becomes disabled either if a task fails or if the user requests the workflow to be paused. No new tasks are initiated for disabled workflows. A resume request from the user is required to make a disabled workflow running again. If the workflow becomes disabled due to task failure, the user must modify the workflow (to correct the failed task) before issuing the resume request.

3.2 Architecture of the Workflow Engine

The workflow service implements operations to (i) create a new workflow, (ii) suspend the execution of a workflow, (iii) resume execution of a workflow, (iv) modify a workflow, and (v) cancel a workflow. The service also supports monitoring of workflows, either by explicit status requests or by asynchronous notifications of updates. To support a wide range of client requirements, different granularities of notifications are available, ranging from a single message upon workflow completion to detailed updates every time a task changes its state. As Java WS Core contains mechanisms for managing WS-Resources (in this case workflows), the monitoring functionality as well as operations (iv) and (v) are trivial to implement (using WS-Notifications [8], and WSRF [13], respectively).

The architecture of the workflow engine is shown in Figure 1. User credentials are delegated from clients to the workflow service to be used when interacting with Grid resources. This requires the *Web service interface* to perform authentication and authorization of clients. All incoming requests are forwarded to the *Coordinator*, which organizes and manages the execution of tasks (and subworkflows) in the workflow and handles workflow state. When a new workflow is requested, the Coordinator uses the *Input Converter* plugin(s) to translate the input workflow description from the native format specified by the client to the internal workflow language. However, the Input Converters do typically not translate the individual task descriptions, as these are only to be read by the *Grid Executor* plugin(s), which the Coordinator invokes to process one (or more) tasks.

The Grid Executor interface defines operations to initiate new tasks, to reconnect to already initiated tasks after service restart, and to cancel tasks, corresponding to the create, resume and cancel operations in the Web service interface. There is however no operation to pause a running task, as this functionality generally is not supported by Grid middlewares. Computational Grid Executors

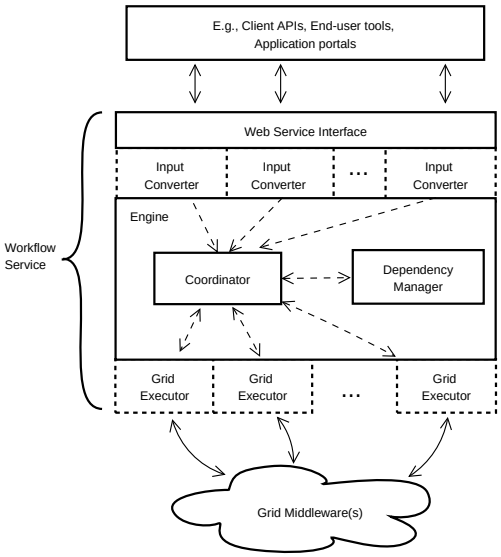


Fig. 1. Overview of the workflow service architecture.

also ensure that tasks' input and output files are transferred in compliance with the data dependencies in the workflow, but are unaware of the context (the workflow) to which each task belongs. This type of Grid Executor only requires a basic job submission mechanism, e.g., WS-GRAM [7], but can also make use of sophisticated frameworks, e.g., the GJMF [3] for resource brokering and fault tolerant job execution, should such functionality be available. Scheduling is performed on a per-task basis by the Grid Executors plugins. However, tools for planning or pre-scheduling of workflows (e.g., Pegasus [2]) can be employed if such functionalities are required. Moreover, support for abstract and concrete workflows is granted via the Executor plugins and external tools respectively.

Before the Coordinator can invoke the Grid Executor(s) in order to start new tasks, the *Dependency Manager* is used to select which task(s) to execute. This module keeps track of dependencies between tasks (and subworkflows) in a workflow, and determines when a task (or subworkflow) is ready to start. The Coordinator invokes the Dependency Manager to get a list of tasks available for execution when a new workflow is started, when a task in an existing workflow completes, and when a paused workflow is resumed.

3.3 Properties of the Workflow Language

In the workflow service, workflows are described in a data flow language, defined using XML schema. In this language, users specify task dependencies, not task execution order. This removes the burden of figuring out which tasks can execute in parallel as this is the responsibility of the Dependency Manager.

The workflow language supports arbitrary nesting of tasks and subworkflows within a workflow. Each task (or workflow) specifies a set of input and output

ports. A (sub)workflow contains a set of *links*, where each link connects an output port of one task/workflow with an input port of another. The task description contains a field to specify how to perform the task. By having this field generic, the usage of multiple Grid task description formats is possible. Different formats for individual task descriptions may even be used within the same workflow. This design also enables support for new task types, e.g., database queries and Web service invocations, to be added by implementing Grid Executor plugins rather than extending the workflow language.

4 Analysis and Comparison with other Systems

One of the main objectives of this work is to provide independence not only of Grid middleware but also of input representation, the latter achieved by converters that translate different workflow languages to the service's internal data flow language. How difficult these translations are depend on the style of the original client's language and the amount and type of information that can be expressed in that language. Data flow languages with similar input/output port structure are simple to translate. *Control flow* languages can also be translated by specifying ports that represent flow of control rather than data transfers. For example, the subset of the Karajan language [16] that performs basic interactions with Grid resources (job submissions, file transfers, and sequential and parallel definition of tasks) has been translated as described above.

Petri net languages pose more difficulties. Places and transitions representing data flows can easily be translated to the service's internal data flow language. However, there is not an equivalent concept for representing loops in the service's language. Finally, it can also be hard to translate languages that do not have all the information encoded in the workflow description but rely on the runtime system to obtain the missing information (e.g., a workflow system that dynamically queries a repository to obtain the input/output structure of workflow tasks).

While several workflow projects have been built to interact with Grid systems [15, 14, 1], many of them have not been designed for exclusive use of Grid resources for workflow execution. Nevertheless they are integrated solutions with sophisticated graphical environments, workflow repositories, and fault management mechanisms. Our work does not attempt to replace those systems, but to provide a means for accessing advanced capabilities offered by multiple Grid middlewares. These benefits are obtained by the separation of the workflow execution from its definition and by making use of well-established protocols. Furthermore, implementing the workflow engine as a stateful WSRF service facilitates the management and control (including fault recovery) of long-running workflows which are common in Grid computing.

The P-GRADE portal [12] and Karajan [16] also focus on the use of resources from different Grids within the same workflow. P-GRADE offers a collaborative environment in which multiple users define workflows through a client application, and control and manage workflows through a portal. The workflows can access resources from multiple Globus-based virtual organizations. Our work

goes beyond this functionality by adding the capability of using other middlewares besides Globus and also offering independence of input language. Karajan also provides a level of interoperability between different execution mechanisms (mainly GT2, GT4, Condor, and the SSH protocol) through the use of providers that allow selection of middleware at runtime. However, while Karajan has a stronger focus on the interaction between users and workflows, our work focuses on handling the workflow state, delegating the interaction with users to clients that have access to the workflow service.

There are a few projects that are using WSRF to leverage the construction of workflow services. The Grid Workflow Execution Service (GWES) [11] uses a Petri net language to define and control Grid workflows. Besides the differences in workflow language type, the main difference between GWES and our work is the ability of using multiple input representations offered by our contribution. The Workflow Enactment Engine Project (WEEP) [17] provides a BPEL engine for Grid Workflows. The engine is accessible as a WSRF service running in a GT4 container. However, WEEP is focused on Web service invocations and not on interfacing with Grid middleware.

5 Concluding Remarks

The goal of this research is to investigate how to design a light-weight workflow engine that can be reused by different high-level tools. General requirements for portability and interoperability are supported by the use of an appropriate plugin-structure for workflow language formats and for interacting with different Grid middlewares. Scalability is obtained by handling multiple workflows and by supporting large hierarchical workflows. The workflow service performs monitoring, state management, fault recovery, and it uses appropriate security mechanisms to achieve user isolation. The Executor plugins handle data movement, job submission, information retrieval, and just-in-time scheduling. External tools can be employed for planning and pre-scheduling of workflows. We finally note that much of the supported functionality is obtained with little or no effort by appropriate use of the WSRF.

6 Acknowledgements

We thank P-O Östberg for fruitful discussions on workflow system design and language constructs, and for collaboration in the integration of the GJMF [3]. We are also grateful to the anonymous referees for their constructive comments.

References

1. 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, *Intl. Workshop on Scientific Applications on Grid Computing (SAG'04),* LNCS 3458, pages 119–132. Springer-Verlag, 2005.

2. 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.
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, 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).
5. 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*, pages 212–220. IEEE CS Press, 2005.
6. S. Eswaran, D. Del Vecchio, G. Wasson, and M. Humphrey. Adapting and evaluating commercial workflow engines for e-Science. In *Second IEEE International Conference on e-Science and Grid Computing*. IEEE CS Press, 2006.
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,* LNCS 3779, pages 2–13. Springer-Verlag, 2005.
8. 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, May 2007.
9. 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.
10. 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.
11. A. Hoheisel. User tools and languages for graph-based Grid workflows. *Concurrency Computat.: Pract. Exper.*, 18(10):1101–1113, 2006.
12. P. Kacsuk and G. Sipos. Multi-grid and multi-user workflows in the P-GRADE Grid portal. *J. Grid Computing*, 3(3-4):221–238, 2006.
13. OASIS. OASIS Web Services Resource Framework (WSRF) TC. http://www.oasis-open.org/committees/wsrf/, May 2007.
14. 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.
15. I. Taylor, M. Shields, I. Wang, and A. Harrison. The Triana workflow environment: architecture and applications. In I. Taylor et al., editors, *Workflows for e-Science*, pages 320–339. Springer-Verlag, 2007.
16. G. von Laszewski and M. Hategan. Workflow concepts of the Java CoG Kit. *J. Grid Computing*, 3(3–4):239–258, 2005.
17. WEEP. The Workflow Enactment Engine Project. http://weep.gridminer.org, May 2007.
18. J. Yu and R. Buyya. A taxonomy of workflow management systems for Grid computing. *J. Grid Computing*, 3(3–4):171–200, 2006.

VI

Paper VI

Combining Local and Grid Resources in Scientific Workflows (for Bioinformatics)*

A.C Berglund[1], E. Elmroth[2], F. Hernández[2],
B. Sandman[2], and J. Tordsson[2]

[1] *The Linnaeus Centre for Bioinformatics,*
Uppsala University, SE-751 24 Uppsala, Sweden
ann-charlotte.sonnhammer@lcb.uu.se

[2] *Dept. of Computing Science and HPC2N,*
Umeå University, SE-901 87 Umeå, Sweden
{*elmroth, hernandf, c01bsn, tordsson*}*@cs.umu.se*

Abstract: We examine some issues that arise when using both local and Grid resources in scientific workflows. Our previous work addresses and illustrates the benefits of a light-weight and generic workflow engine that manages and optimizes Grid resource usage. Extending on this effort, we here illustrate how a client tool for bioinformatics applications employs the engine to interface with Grid resources. We also explore how to define data flows that transparently integrates local and Grid subworkflows. In addition, the benefits of parameter sweep workflows are examined and a means for describing this type of workflows in an abstract and concise manner is introduced. Finally, the above mechanisms are employed to perform an orthology detection analysis.

Key words: Grid, workflow, data flow, parameter sweep, bioinformatics

* By permission of Springer-Verlag

Combining Local and Grid Resources in Scientific Workflows (for Bioinformatics) *

Ann-Charlotte Berglund[1], Erik Elmroth[2], Francisco Hernández[2], Björn Sandman[2], and Johan Tordsson[2]

[1] The Linnaeus Centre for Bioinformatics, Uppsala University
ann-charlotte.sonnhammer@lcb.uu.se, www.lcb.uu.se
[2] Department of Computing Science and HPC2N, Umeå University
{elmroth, hernandf, c01bsn, tordsson}@cs.umu.se, www.gird.se

Abstract. We examine some issues that arise when using both local and Grid resources in scientific workflows. Our previous work addresses and illustrates the benefits of a light-weight and generic workflow engine that manages and optimizes Grid resource usage. Extending on this effort, we here illustrate how a client tool for bioinformatics applications employs the engine to interface with Grid resources. We also explore how to define data flows that transparently integrates local and Grid subworkflows. In addition, the benefits of parameter sweep workflows are examined and a means for describing this type of workflows in an abstract and concise manner is introduced. Finally, the above mechanisms are employed to perform an orthology detection analysis.

Key words: Grid, workflow, data flow, parameter sweep, bioinformatics

1 Motivation and Background

To date many available tools (e.g., [1–3]) for scientific workflows have sophisticated functionality that enables, e.g., design, enactment, and scheduling of workflows, while providing support for data management and fault tolerance. However, most of these tools have at least one of three shortcomings: (1) the tool is a monolithic application where the above listed functionality cannot easily be extracted and reused, (2) the tool targets a specific scientific domain and to adapt it to new domains is a major, if not impossible, endeavor, and (3) the tool is focused on workflow execution on client machines ignoring the benefits of using Grid resources for computationally or data intensive tasks.

In previous work [4, 5] we have discussed these problems and argued in favor of a loosely coupled design where workflow capabilities are exposed as a composable set of workflow services with clear separation of concerns. By orchestrating these services, environments tailored to the needs of a certain group of users

* 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.

can be constructed with much less effort compared to a (re)implementation of a full-featured system. Our design and implementation of the Grid Workflow Enactment Engine (GWEE) [4] demonstrates the viability of this approach.

This work is a direct continuation of these efforts. We identify the following five contributions, each described in a separate section of the paper. First, we illustrate the feasibility of the concept of a composable set of workflow capabilities by demonstrating how a client tool, adapted for use within bioinformatics, can be built on top GWEE. The problems of decoupling data from task specification and that of dynamically adjusting workflow tasks during runtime are both addressed with task templates that delay the binding of task parameters until workflow execution. We then examine data flows integrating the client machine and Grid resources. We also present a mechanism to simplify parameter sweep studies. Finally, these results are used to reduce the complexity of a particular bioinformatics problem - orthology detection analysis.

2 GWEE Details and the Workflow Client Tool

Before discussing the data management issues encountered when combining Grid and local workflows, we give a brief overview of the GWEE and the client tool. The GWEE [4] is a Grid workflow enactment engine that is exposed as a Web service. A strict separation between workflow enactment logic and execution of individual tasks (e.g., computational jobs and file transfers) enables independence of workflow description language and interoperability with multiple Grid middlewares. The GWEE uses a data flow model of computation in which task dependencies define the execution order of the workflow. However, workflows driven (in part or in full) by a control flow are also supported.

While the GWEE provide access to Grid resources, the interaction with users is delegated to a client tool that uses the service interface to start and terminate workflows, as well as pause and resume a running workflow. In this tool workflows can also be designed, in a drag and drop manner, by connecting tasks together in a graph that specifies a control and/or a data flow. The client can handle mixed local and Grid workflows. Simple tasks that are not computationally intensive, e.g., preparation of input files, are typically performed on the client side whereas long running jobs preferably should be combined into a subworkflow that can be sent to the GWEE for enactment on the Grid. The state of (running) workflows can be stored to, and loaded from disk. The client can hence be disconnected from the GWEE during the execution of a long Grid subworkflow. This is useful, e.g., when running the client on a laptop during travel or for Grid workflows executing for days or weeks. The client tool can be updated of Grid subworkflow progress either by synchronous requests or by notifications.

In both the GWEE and the client tool, workflow tasks consist of three parts (1) a set of input ports, (2) a set of output ports, and (3) a process that maps the input ports to the output ports (i.e., a job that receives inputs and produces outputs). Links from output ports to input ports define task dependencies. Similarly to other data flow approaches [6], the links are associated with tokens

that are moved between tasks. For the GWEE these tokens represent pointers to data files with the transfer medium dependent upon the supported infrastructure (e.g., GridFTP) and the transfer carried out by middleware specific plugins. On the client side, a token holds the name of a locally stored file that can be accessed from both tasks in a producer/consumer fashion.

3 Task templates

Scientific workflows running on the Grid typically consist of self-contained command line applications or scripts. Execution is usually configured via environment variables or command line arguments and all communication, in and out, is performed via data files or the standard streams (i.e., *stdin*, *stdout*, and *stderr*). While this task configuration information must be explicitly described prior to submission to Grid resources, it is not necessary when designing the workflows. The binding of particular parameters can be delayed until workflow enactment, which is beneficial since the required information is not always known at design time. In addition, to support non-deterministic[3] workflows, parts of the workflow description must be completed during enactment with values produced as output from (previously) executed workflow tasks.

The use of command line applications introduces the problem of embedding the required task information into the workflow model. For example, as mentioned in the previous section, the GWEE follows a data flow approach in which all information is moved through ports. Values passed through these ports (e.g., name of input files) are required to enact the workflow and correspond to the arguments and data products of the command line applications. This introduces the difficulty of mapping the arguments to the corresponding input and output ports for the correct arrangement of dependencies delineating the flow of data.

An often employed solution is to use wrappers to either include individual applications into the workflow model or to execute those applications in a Grid [7–9]. GEMLCA [7] exposes a generic service layer to execute command line applications without the need of re-engineering the application. The Styx Grid Services [8] create services of command-line applications and allows them to run remotely as if they were local programs. In Styx, shell scripts can be used to compose the services as workflows. These solutions enable the reuse of the applications with different data [10] but require a service for each individual application. If services are used only for a short period of time, these solutions become too heavyweight as the overhead of service creation outweighs the execution time of the wrapped application. In non-Grid environments, workflow tools [1–3] usually offer a framework with a set of functionalities that can be extended by writing framework specific applications using libraries developed exclusively for this purpose [11]. However, when moved to the Grid this solution is not always feasible as the application source code need not be accessible.

[3] By the term *non-deterministic*, we refer to a workflow where the precise enactment is not known in advance and it is determined dynamically during execution. *Dynamic* workflows, in contrast, are actively steered by users.

Compared to the wrapper solution, our approach is lightweight as it involves only the specification of a parameterized *task-template* where the workflow designer specifies the correct mapping of input and output ports to the parameters of the application. The designer can also specify which arguments are to be determined at a later stage (during enactment). A task-template is an incomplete task description, in JSDL [12], that contains name-value mappings for missing task parameters. The template is transformed into an executable task once the values for these missing parameters are available. The actual values for the parameters are defined (and updated) using the task ports.

By filling out a single form that is automatically generated from the task template, users can specify values for the template parameters without being concerned about low-level details such as the syntax of the workflow and/or task descriptions. The template functionality also allows the transformation of the task outline to the resulting task to be delayed until task execution time, and hence enables non-deterministic workflows.

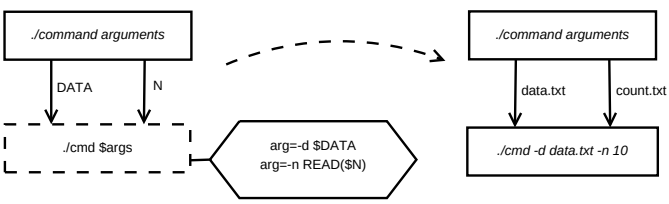


Fig. 1. Basic operation of the template, including usage of the read operation.

For some task templates, the value specified in the input port is not sufficient information for the workflow to be enacted correctly. Instead, the data value(s) produced at that port is required. To access these values, we provide a *read* operation that uses the port content instead of the port name for task generation in the template. The POSIX command *xargs* provides a functionality similar to the read task, but is restricted to mapping stdin to command line arguments.

A simplified illustration of the template functionality, including the read operation, is shown in Figure 1. In this figure, the incomplete task description (dashed box) contains a template (shown as a hexagon). After receiving values (data.txt, count.txt) for the template parameters ($FILE, $N), the task outline is transformed to the executable task, shown to the right. Note how the read operation enables the content (10) of the port marked N to be used instead of the name of the token passed on that port (count.txt).

4 Transparent Local and Grid Data Flows

Data transfers become cumbersome for data flow workflows that contain both local and Grid subworkflows. Special care is required to ensure that the output of a local task is available as input to a subsequent Grid task. To handle this, the

workflow client explicitly transfers the data to a Grid-available storage element and then back to the client machine once the Grid subworkflow is completed. These transfers, being control flow activities, are hidden from the user and the visual workflow displayed in the client tool remains pure data flow.

Input ports can be used not only to define input from previous tasks in the workflow, but also to specify dependencies on external data. In Grid subworkflows, this feature is utilized to automatically transfer required files, e.g., binaries and other task input files not originally produced by the workflow.

5 Parameter Sweeps in Workflows

A commonly used pattern in workflows is parameter sweeps, where one computational task (or a subworkflow) is executed with different data. Parameter sweep require a suitable partition of the input data where each partition is assigned to a different task (or subworkflow). Embarrassingly parallel problems such as parameter sweep studies are well suited to Grid environments, where large numbers of resources are available. Even though parameter sweep workflows can be done without Grid workflows, data management in the client is simplified when the parameter sweep can be treated as one (Grid) subworkflow, instead of a (potentially large) set of individual tasks. This increased abstraction is beneficial both from a usability and a performance point of view. Other advantages include the reuse of a familiar programming paradigm akin to map-reduce or scatter-gather reductions. By implementing parameter sweeps as a workflow graph rewrite on the client side, iterations of the *parallel-for* style are possible, regardless of whether or not iterations are supported by the workflow engine.

In data flow languages, parameter sweeps are typically implemented using either graph rewrites, see e.g., Triana [3] and Pegasus [13] or with higher-order functions as done, e.g., by Kepler [1]. Generation of the vector to sweep over can be done statically before the execution as in Triana [14], or dynamically during workflow enactment, the latter approach used by Askalon [15], Kepler [1], and Taverna [2]. The P-GRADE [10] data composition strategies allow a custom pairing of service input ports, typically through dot or cross product. Pautasso et al [16] study parameter sweeps and other parallel patterns for workflows.

Our implementation of parameter sweeps is based on a graph rewrite that can take place either before or during workflow execution. A parameter sweep consists of three parts: i) a *divider* that generates the desired partition, ii) the task template(s) forming the subworkflow that should be swept over, and iii) a *merger* that collects the results. The divider and the corresponding merger are computation tasks in the workflow. The rationale behind this is that the workflow designer knows best how to properly generate the parameter sweep instances, i.e., the branches in the parameter sweep. By allowing the designer to implement the merger and divider (typically done in a Turing-complete scripting language) rather than to choose from a few predefined partition types, new parameter distribution patterns can be added without modifying the workflow language. Commonly used patterns include for each input file, for each part of a large

file (that is to be split up), and Cartesian product between two sets, the latter illustrated in the bioinformatics use case in Section 6.

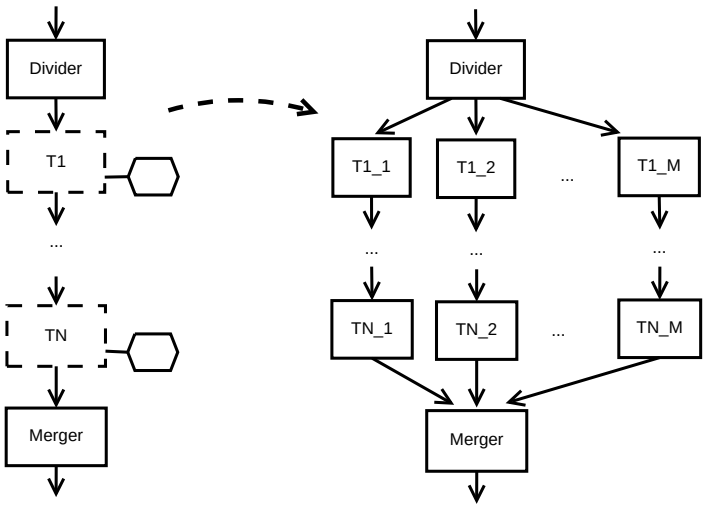


Fig. 2. Expansion of a subworkflow into a parameter sweep.

The divider task output generates the desired data partition for the parameter sweep. Each line of output contains template value bindings for one branch in the parameter sweep. When the divider task completes, the template subworkflow is rewritten (expanded) into a task subworkflow, that can be executed either locally or on the Grid. The divider task can either be executed statically before the workflow is started or dynamically during workflow enactment. In the latter case, the divider itself can contain templates that allow information collected during workflow execution guide the generation of sweep instances. The merger is always executed during workflow run time. It can be configured either to be part of the generated (Grid) subworkflow or be executed locally.

An illustration of the parameter sweep expansion process is shown in Figure 2. From the M lines of divider task output, the incomplete task descriptions (dashed boxes T1 ... TN) with their corresponding templates (depicted as hexagons) are resolved into sweep instances, each with concrete tasks. The parameter sweep in Figure 2 is static as the divider itself does not make use of the template functionality to determine M during workflow enactment. For clarity, the figure shows a set of tasks connected as a pipeline, but the parameter sweep functionality can be applied to any connected subworkflow.

6 InParanoid - a Bioinformatics Use Case

With the new sequencing technologies [17], the bioinformatics field is facing an avalanche of sequence data to be analyzed. As many of the computational problems are embarrassingly parallel, there is a growing interest in the bioinformatics field for Grid techniques in order to accelerate the data processing.

The analysis of biological data is typically a mix of short processing steps and larger computations that can easily be parallelized. These steps are all part of a scientific discovery workflow for answering a biological question. Therefore, for a workflow tool to be useful in the bioinformatics field it must have the ability to combine local and Grid subworkflows into a larger workflow. The large-scale bioinformatics analyzes are made on general purpose computational Grids, such as EGEE and NorduGrid, and can hence not rely on pre-installed software. All this considered, these analyzes are today not done using a workflow tool. Instead, the different tasks in the analysis are typically connected by Perl scripts and are manually submitted to the computational Grids. The drawbacks with this approach are that (1) it can be difficult to exactly replicate the analysis, (2) there is no easy way to reuse the scientific discovery workflow, and (3) it is difficult to analyze and verify this workflow.

Here we have implemented a workflow for the orthology detection tool InParanoid [18]. Orthology and paralogy are key concepts in comparative genomics, and both refer to genes that are related through common descent, i.e., genes that are homologous. In the former case through speciation and in the latter through duplication. Due to the different evolutionary relationships, orthologous genes are more likely to have preserved the biological role than paralogous genes. Therefore, it is important to be able to distinguish between these two types of homologous genes during, for example, the functional annotation of newly sequenced genomes. The data used here are the collected protein sequences, the so-called proteomes, from five species[4] Candida glabrata (NCBI), Drosophila pseudoobscura (FlyBase), Escherichia coli K12 (NCBI), Kluyveromyces lactis (NCBI), and Saccharomyces cerevisiae (SGD).

The InParanoid workflow consists of four steps: (1) format gene databases and filter the proteomes to get the longest transcript per gene, (2) run the Basic Local Alignment Search Tool (BLAST) [19] on all pairs of proteomes including self-against-self, (3) filter the BLAST results, and (4) run the InParanoid application. These four steps are illustrated in Figure 3. Step 1 is a foreach-style parameter sweep that executes locally. The Grid executed parameter sweep in Step 2 performs sequence similarity searches using the BLAST tool. Step 3 is a local parameter sweep and Step 4 is a Grid subworkflow. The parameter sweeps in steps 2 and 3 generate sweep instances according to a Cartesian product pattern. The InParanoid workflow makes extensive use of the template functionality, e.g., to pass genome file names and gene transcript lengths to the command line applications. The edges in the resulting workflow graph in Figure 3 depict a simplified control flow. The complete workflow combines local control flow and Grid data flow, the latter used to install the binaries (BLAST and InParanoid) and to transfer the gene databases and results.

The InParanoid workflow was run on a dedicated Grid testbed with four resources that run Ubuntu Linux 2.6.24, NorduGrid/ARC 0.6, Maui 3.2.6, and Torque 2.1.9. Each resource had one HP DL165 G5 Opteron 2346 HE, with a

[4] NCBI, FlyBase, and SGD refer to the respective databases where the proteome data are publicly available.

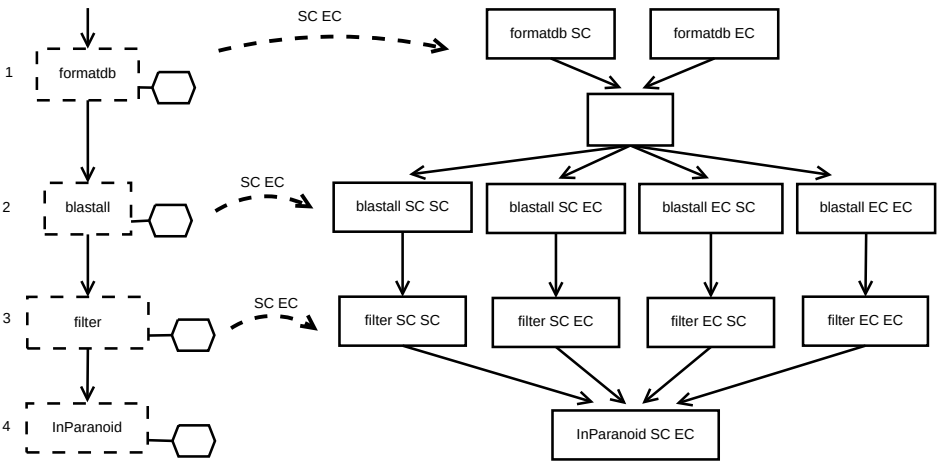


Fig. 3. A simplified view of the templates (left) and the resulting InParanoid workflow (right) instantiated for two proteomes (SC and EC). The numbers $1, \ldots, 4$ correspond to the four steps described in Section 6.

quad core 1.8 GHz CPU and 4GB of memory as backend. The local steps in the workflow were enacted on Ubuntu Linux 2.6.27 laptop with an Intel 1.2GHz Core 2 Duo CPU and 4 GB memory. A 100 Mbit/s network connected all machines.

Table 1 shows performance results for pairwise InParanoid comparisons of the five investigated proteomes: Drosophila pseudoobscura (DP) with 9871 protein sequences, Saccharomyces cerevisiae (SC) with 5792 sequences, Kluyveromyces lactis (KL) with 5336 sequences, Candida glabrata (CG) with 5192 sequences, and Escherichia coli K12 (EC) with 4243 sequences.

Notably, as exactly the same workflow is enacted when using one and four Grid resources and Step 2 is the only parallel part, the two runs give similar results for all other columns in Table 1. The runtime complexity of BLAST is $\mathcal{O}(mn)$, where m and n are the sizes of the compared proteoms. The obtained speedup hence varies with the proteom size induced load (im)balance of Step 2. Notably, the varying from one to four resources only applies to the Grid workflow in Step 2, although the speedup is calculated for the whole workflow. The last column of Table 1 shows that the parallel part (Step 2) constitutes between 85 and 95 per cent of the total execution time. We remark that 90 per cent parallel execution time would, with perfect load balance and neglectable overhead, result in a speedup of 3.08 for four resources. In traditional parallel applications that execute in a cluster and communicate through message passing, poor speedup due to load imbalance results in idle nodes and wasted CPU cycles. On the contrary, in the InParanoid workflow, there is no waste of resources, as cluster nodes can be used by other tasks once the shorter BLAST jobs in Step 2 completes. Notably, the parallel speedup of the InParanoid workflow could be improved with a finer-grained data distribution pattern for Step 2, e.g., by partitioning the proteom files into equally sized chunks. We however remark that the above

Table 1. Performance results for the InParanoid workflow for proteom pairs as given in the first column. The columns labeled 1 to 4 shows the time for workflow steps 1 to 4. Next follows the overhead (which is also part of the time in the column labeled 2) for file transfer to and from the Grid, all other workflow enactment overhead, the makespans of the complete workflow enacted with four and one Grid resources, and the associated speedup (calculated as MS(1)/MS(4)), respectively. The last column shows how large part Step 2 (the part to be executed in parallel) constitutes of the makespan for the one resource case. All time units are in seconds.

Proteoms	1	2	3	4	File staging	OH	MS(4)	MS(1)	Speedup	T_p(%)
DP-SC	2	5639	340	737	152 (1581MB)	19	6737	14213	2.11	92.2
DP-KL	2	5606	297	799	134 (1381MB)	17	6720	12894	1.92	91.5
DP-CG	2	5695	321	768	147 (1503MB)	20	6806	13590	2.00	92.0
DP-EC	1	5581	189	219	92 (899MB)	15	6005	8408	1.40	94.9
SC-KL	1	2256	188	814	94 (922MB)	19	3278	8098	2.47	87.4
SC-CG	1	2226	226	1043	110 (1096MB)	18	3514	9304	2.65	85.9
SC-EC	1	2233	85	219	53 (436MB)	17	2555	4044	1.58	93.3
KL-CG	1	1948	172	768	89 (844MB)	18	2907	7599	2.61	87.2
KL-EC	1	1567	58	189	41 (307MB)	15	1830	3258	1.78	91.5
CG-EC	1	1891	74	204	47 (383MB)	16	2186	3647	1.67	92.4

tests are an illustration of the functionality of the parameter sweep mechanism, rather than an attempt to optimize the performance of the InParanoid workflow.

7 Conclusions and Future Work

We demonstrate how data flows seemlessly can integrate local and Grid resources. We also introduce a more flexible parameter sweep tool than those available in current workflow systems. Our task template mechanism enables non-deterministic workflows and delineates data flow for command line applications. Through a combination of these mechanisms, we illustrate how to simplify incorporation of Grid resources in the InParanoid workflow. Future work includes analyzing conceptual interoperability aspects for scientific workflows to lay a foundation for workflow reuse and workflow system interoperability.

References

1. Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Leen, E.A., Tao, J., Zhao, Y.: Scientific workflow management and the Kepler system. Concurrency Computat.: Pract. Exper. **18**(10) (2006) 1039–1065
2. Oinn, T., Addis, M., Ferris, J., Marvin, D., Senger, M., Greenwood, M., Carver, T., Glover, K., Pocock, M., Wipat, A., Li, P.: Taverna: A tool for the composition and enactment of bioinformatics workflows. Bioinformatics **20**(17) (2004) 3045–3054
3. Taylor, I., Shields, M., Wang, I., Harrison, A.: The Triana workflow environment: architecture and applications. In Taylor, I., et al., eds.: Workflows for e-Science. Springer (2007) 320–339

4. Elmroth, E., Hernández, F., Tordsson, J.: A light-weight Grid workflow execution engine enabling client and middleware independence. In Wyrzykowski, R., et al., eds.: Parallel Processing and Applied Mathematics, LNCS 4967, Springer 754–761
5. Elmroth, E., Hernández, F., Tordsson, J., Östberg, P.O.: Designing service-based resource management tools for a healthy Grid ecosystem. In Wyrzykowski, R., et al., eds.: Parallel Processing and Applied Mathematics, LNCS 4967, Springer-Verlag 259–270
6. Johnston, W.M., Hanna, J.R.P., Millar, R.J.: Advances in dataflow programming languages. ACM Computer Surveys **36**(1) (2004) 1–34
7. Delaitre, T., Kiss, T., Goyeneche, A., Terstyanszky, G., Winter, S., Kacsuk, P.: GEMLCA: Running legacy code applications as grid services. J. Grid Computing **3**(1–2) (2005) 75–90
8. Blower, J.D., Harrison, A.B., Haines, K.: Styx Grid Services: Lightweight middleware for efficient scientific workflows. Scientific Programming **14**(3–4) (2006) 209–216
9. Wang, I., Taylor, I., Goodale, T., Harrison, A., Shields, M.: gridMonSteer: Generic Architecture for Monitoring and Steering Legacy Applications in Grid Environments. In Cox, S., ed.: The UK e-Science All Hands Meeting 2006. (2006) 769–776
10. Glatard, T., Sipos, G., Montagnat, J., Farkas, Z., Kacsuk, P.: Workflow-level parametric study support by MOTEUR and the P-GRADE portal. In Taylor, I., et al., eds.: Workflows for e-Science. Springer (2007) 279–299
11. Huang, Y., Taylor, I., Walker, D.W., Davies, R.: Wrapping legacy codes for grid-based applications. In: International Parallel and Distributed Processing Symposium (IPDPS'03), IEEE Computer Society (2003) 139–145
12. Anjomshoaa, A., Brisard, F., Drescher, M., Fellows, D., Ly, A., McGough, A.S., Pulsipher, D., Savva, A.: Job Submission Description Language (JSDL) specification, version 1.0 http://www.ogf.org/documents/GFD.56.pdf, February 2009.
13. Gil, Y., Ratnakar, V., Deelman, E., Mehta, G., Kim, J.: Wings for Pegasus: Creating large-scale scientific applications using semantic representations of computational workflows. In Cheetham, W., Goker, M., eds.: 19th Conference on Innovative Applications of Artificial Intelligence (IAAI). (2007) 1767–1774
14. Churches, D., Gombas, G., Harrison, A., Maassen, J., Robinson, C., Shields, M., Taylor, I., Wang, I.: Programming scientific and distributed workflow with Triana services. Concurrency Computat.: Pract. Exper. **18**(10) (2006) 1021–1037
15. Fahringer, T., Prodan, R., R.Duan, Hofer, J., Nadeem, F., Nerieri, F., Podlipnig, S., Qin, J., Siddiqui, M., Truong, H.L., Villazon, A., Wieczorek, M.: ASKALON: A development and Grid computing environment for scientific workflows. In Taylor, I., et al., eds.: Workflows for e-Science. Springer (2007) 450–471
16. Pautasso, C., Alonso, G.: Parallel computing patterns for grid workflows. In: Proc. of the HPDC2006 Workshop on Workflows in Support of Large-Scale Science (WORKS06) Paris France. (June 2006)
17. Hall, N.: Advanced sequencing technologies and their wider impact in microbiology. J. Exp. Biol. **9**(210) (2007) 1518–1525
18. Remm, M., Storm, C.E.V., Sonnhammer, E.L.L.: Automatic clustering of orthologs and in-paralogs from pairwise species comparisons. J. Mol. Biol. **314** (2001) 1041–1052
19. Altschul, S.F., Madden, T.L., Schäffer, A.A., Zhang, J., Zhang, Z., Miller, W., Lipman, D.J.: Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Res. **17**(25) (1997) 3389–3402

VII

Paper VII

Three Fundamental Dimensions of Scientific Workflow Interoperability: Model of Computation, Language, and Execution Environment*

E. Elmroth, F. Hernández, and J. Tordsson

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

Abstract: We investigate interoperability aspects of Grid workflow systems with respect to *models of computation (MoC)*, *workflow languages*, and *workflow execution environments*. We focus on the problems that affect interoperability and illustrate how these problems are tackled by current scientific workflows as well as how similar problems have been addressed in related areas. Emphasis is given to the differences and similarities between local and Grid workflows and how their peculiarities have a positive or negative impact on interoperability. Our long term objective is to achieve (*logical*) interoperability between workflow systems operating under different MoCs, using distinct language features, and using different execution environments.

* UMINF 09.05

Three Fundamental Dimensions of Scientific Workflow Interoperability: Model of Computation, Language, and Execution Environment

Erik Elmroth Francisco Hernández Johan Tordsson

UMINF 09.05

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

Abstract

We investigate interoperability aspects of Grid workflow systems with respect to *models of computation (MoC)*, *workflow languages*, and *workflow execution environments*. We focus on the problems that affect interoperability and illustrate how these problems are tackled by current scientific workflows as well as how similar problems have been addressed in related areas. Emphasis is given to the differences and similarities between local and Grid workflows and how their peculiarities have a positive or negative impact on interoperability. Our long term objective is to achieve (*logical*) interoperability between workflow systems operating under different MoCs, using distinct language features, and using different execution environments.

1 Introduction

To date, scientific workflow systems offer rich capabilities for designing, sharing, executing, monitoring, and overall managing of workflows. The increasing use of these systems correlates with the simplicity of the workflow paradigm that provides a clear-cut abstraction for coordinating stand-alone activities. With this paradigm scientists are able to concentrate on their research at the problem domain level without requiring deep knowledge of programming languages, operating systems, arcane use of libraries, or hardware infrastructure. In addition, the ease by which scientists can describe

experiments, share descriptions and results with colleagues, as well as automate the recording of vast amounts of data, e.g., provenance information and other data relevant for reproducing experiments, have made the workflow paradigm the fundamental instrument for current and future scientific collaboration.

Currently, there are many sophisticated environments for creating and managing scientific workflows that have lately also started to incorporate capabilities for using powerful Grid resources. Although similar in many respects, including domains of interest and offered capabilities, existing workflow systems are not yet interoperable. Rather than discussing if workflow systems are completely interoperable or not at all, here we argue that interoperability between workflow systems must be considered from three distinct dimensions: *model of computation (MoC)*, *workflow language*, and *workflow execution environment*. In previous work [20] we have demonstrated workflow execution environment interoperability by showing how a workflow tool can interoperate with multiple Grid middleware. Extending on this effort, we here discuss interoperability at the other two dimensions, i.e., MoC and workflow language.

With this paper our contributions are the following. We start a dialogue and argue that we must change the manner in which interoperability has been addressed and start concentrating on a three dimensional model that considers interoperability from the MoC, the language, and the execution environment dimensions. We present the reasons why dataflow networks have not been used in Grid settings and introduce some adaptations required in this MoC to be able to make use of it in Grid workflows. We further investigate the minimum language constructs required for a language to be expressive enough for supporting scientific workflows. We argue for a distinction between the languages used for describing complex workflows executing on the end-users desktop machine (i.e., *local workflows*) and possibly simpler languages only used for coordinating computationally intensive sub-workflows that run on the Grid. We support our discussion on results and algorithms from the areas of theory of computation, compiler optimization, and (visual) programming language research. With these results as a starting point we discuss language aspects relevant for local and Grid workflows, including iterations, conditions, exception handling, and implications of having a type system associated to the workflow language. We conclude our work in workflow languages by studying the repercussions of choosing between *control-driven* and *data-driven* style of representing workflows, including methods for converting between both representations.

The rest of the paper is organized as follows. Section 2 discusses the

current state of the art in scientific workflows, investigates the reasons why interoperability in workflows is desired, and introduces the three dimensions that must be considered when discussing interoperability of scientific workflows. Section 3 introduces some fundamental workflow concepts and definitions that are used throughout the paper. Section 4 discusses issues related to MoCs for Grid workflows including a description of how Petri nets and Coloured Petri nets have been used as MoCs for Grid workflows and a discussion on the reasons why dataflow networks, being the most common MoC for local workflows, has not yet been employed for Grid workflows. Section 5 focuses on workflow language related issues including differences between data-driven and control-driven representations as well as the use and implementation of language constructs such as conditions and iterations in light of their programming languages counterparts. Finally, in Section 6 we discuss how our findings, collectively, have an impact on Grid workflow interoperability and present our concluding remarks.

2 Workflow Interoperability

There are many scientific workflow systems currently in use, e.g., [9, 15, 43, 51]. Several of these have been developed successfully within interdisciplinary collaborations between domain scientists, the *end-users*, and computer scientists, the *workflow engineers*. Some of these systems target a particular scientific domain (e.g., Taverna [51]) while others cover a range of fields (e.g., Triana [9] and Kepler [43]).

The existence of such a wide range of workflow systems is comparable, although to a lesser degree, to the large number of programming languages available. In both cases solutions can be general purpose or tailored for specific domains, and the choice of one over the others depends not only on the problem at hand but also on personal preferences. Moreover, it is not possible to have one solution suitable for all problems and preferred by all users, and it is also not likely for a new solution to emerge and replace all existing ones. Yet, unlike programming languages in which interoperability is achieved at the binary or byte code level and the voices suggesting source-to-source interoperability long faded away, achieving interoperability between workflow systems is a venture of high priority.

In this section we examine motivations for workflow interoperability, introduce a point of view that tackles the problem at multiple dimensions, and investigate various ways in which the topic has been addressed in the literature.

2.1 A case for interoperability

It has recently been suggested [53] that end-users are not really pressing for interoperability among workflow systems. The rationale behind this suggestion is that these systems are developed in tight coordination between end-users and workflow engineers. Hence, instead of using other systems that already offer the required functionalities new features are added when needed. The outcome of this rationale is time consuming as it leads to duplication of efforts, it mis-utilizes resources that could otherwise be employed in more productive endeavors, and it is mainly beneficial for researchers who are involved in this development loop. Yet, it is also the case that users may not be interested in *full interoperability* between workflow systems. Full interoperability is commonly defined as the ability for systems (be them human users or software components) to *seamlessly* use the full functionality provided by the (others) interoperable systems in a totally transparent manner [24]. Deelman et. al. [14] notice that users may want to invoke one workflow system from another one or reuse all or part of a workflow description in a different system. Another motivation for interoperability is due to portability aspects, e.g., due to infrastructure changes. User preferences can also be taken into consideration. Once users become accustomed to a particular system it becomes a costly process to migrate to another one. Although it is possible to enumerate a long list of use cases in which interoperability is of value, we prefer to identify the following two categories that cover several of the use cases according to the purpose for seeking interoperability:

Collaborative and interdisciplinary research. Science is a collaborative endeavor. The importance of current scientific problems have made crucial for these collaborations to become large interdisciplinary enterprises in which scientists from different fields contribute to the final solution. These significant efforts require the sharing of research knowledge, knowledge that is often expressed in workflows.

Several workflow systems have been developed for operation within a specific scientific domain, thus, it is expected that users from different scientific fields, or in some cases from different research groups, use different workflow systems. It is very difficult to change systems just to enable these collaborations. Users are more comfortable working in environments familiar to them and adapting to a different one may involve steep learning curves [31]. For these cases it becomes imperative to coordinate multiple workflow systems within one workflow execution. A typical scenario for this type of interoperability is a workflow

system invoking another one for executing a functionality represented as a sub-workflow.

Lack of capabilities in a workflow system. Adaptation of a system initially designed for one scientific field to fulfill the requirements of another is typically done by extending the set of activities (or capabilities) offered to users, rather than changing the way in which the workflow system itself operates. Such extensions are commonly added by implementing the new functionalities using libraries offered by the systems themselves. However, these functionalities are typically locked-in and can only be used inside the targeted environment making it impossible to share them with users of other systems. It is then important to unlock the functionalities so that they can be used by other systems. The capabilities need not be computations, they can also be support for different hardware or software platforms. An extreme case is illustrated by technology obsolescence. If a system becomes obsolete and needs to be replaced by a newer system, it is of paramount importance to be able to reuse the workflows developed for the older system. This obsolescence is not limited to the workflow system itself but the execution environment, including middleware, as well.

2.2 Multiple dimensions for workflow interoperability

From the two categories enunciated above, we identify three dimensions relevant for scientific workflow interoperability. These dimensions are in line with a previous classification [53], but we argue that some aspects presented in that work, e.g., meta-data and provenance, although very important in practice, are not essential for workflow enactment coordinated by a workflow engine. In practice, a workflow engine can cooperate with a meta-data or provenance manager to achieve other types of interoperability. In this work we focus exclusively in the enactment process, that is, selecting and executing workflow activities free of dependencies (either control or data dependencies). Below we briefly introduce the three dimensions of interoperability and we present further details in the following sections.

2.2.1 Model of computation

The model of computation provides the semantic framework that governs the execution of the workflow, i.e., a MoC provides the rules on how to execute the activities and consume the dependencies. There are many MoCs that

have been considered as central abstractions for coordinating workflows, including Petri nets, dataflow networks, and finite state machines. Some problems are better suited for one MoC and in many cases a single workflow may be required to use abstractions from multiple MoCs.

Strong interoperability in this respect requires the transparent execution of workflows developed for one MoC by another one. A weaker notion is to be able to compose workflows with parts governed by different MoCs. A solution proposed by Zhao et al. [70] uses a bus in which workflow systems are considered as black boxes that can be invoked from other systems. However, for this compositions to be possible the MoCs must be compatible. Compatibility between MoCs has been studied by Goderis et al. [29]. Their work explores when and how different MoCs can be combined. Their contribution is significant but it is focused only on local workflow systems and does not address Grid workflows. Many of the MoCs described by Goderis et al. [29] are not functional in Grid settings as the assumptions of globally controlling coordinators and fine grained token-based interactions between concurrently executing activities are not possible to achieve (yet) in Grid environments. The basis of their work is a hierarchical approach, based on Eker et al. [18], in which workflows from different MoCs can be combined according to the level of the hierarchy. Sub-workflows are then seen as black boxes and their internal MoC is of no importance when working one level above in the hierarchy. Petri nets and DAGs (employed as the structure for specifying dependencies between activities) are the most common MoCs in Grid environments. As such, in Section 4 we look into how Petri nets and Colour Petri nets have been used for Grid workflows. In that section we also investigate the reasons why dataflow networks, the most common MoC for local workflow systems, has not been employed in Grid environments.

2.2.2 Workflow language aspects

One important workflow interoperability aspect is given by the set of supported language constructs. The constructs of interest in this study are iterations and conditions, the latter used both for execution flow control and exception handling. Another related topic is how, if at all, state is modeled in workflows. Our previous work has demonstrated the possibility of completely decoupling the workflow coordination language from the language used to describe individual activities [20]. Others propose to describe workflows with a high-level meta-language, that is not dependent on a particular workflow language [53]. Fernando et al. [23] suggest an intermediate workflow description format, and outline how the languages of Taverna and

Kepler could be represented in such a format. In Section 5 we discuss the trade-offs of having a full featured, Turing complete workflow language versus a simplistic activity coordination language. Furthermore, we investigate the consequences of having a type system attached to a workflow language and discuss the difficulties that the environments of current Grid infrastructures cause in this regard.

2.2.3 Workflow execution environments

In traditional workflow systems, the activities that form the workflow all execute locally on the desktop computer of the user, making it a *local workflow*. The emergence of powerful parallel machines and Grids has opened up the potential for utilizing applications that execute on remote machines and possibly are developed by other scientists or organizations. Accordingly, it is essential for a workflow system to support distributivity at some level. Since many of the current projects are pre-Grid, they, naturally, are not focused on Grid workflows, and are not able to optimize the capabilities that systems designed for Grid usage offer. On the other hand, several of the Grid-only workflow systems can appropriately use Grid resources but they lack the ease of use and facilities offered by the local systems. The necessities of current research demand for a balance between the local and the distributed, so in typical scenarios the local machine is used for menial tasks while Grid resources are used for activities that require extensive use of resources (e.g., computation and storage). In this setting, the benefits of workflow systems is that they abstract the communication complexities required to interact with the Grid.

One issue in the design of toolkits for real-life scientific workflows is the suitable level of granularity for interacting with Grid resources. Some projects, e.g., Kepler [43], Taverna [52], and Karajan [67] use Grid resources on a per-activity basis. Others, e.g., Pegasus [15], GWEE [20], and P-Grade [38] use Grid resources to enact workflows, that typically constitute a computationally intensive subset of a larger workflow. We refer to such sub-workflows as *Grid workflows*. For the rest of this paper, we assume that a workflow is a local workflow that makes use of one or more Grid workflows, the latter ones being our focus.

3 Concepts and Definitions

In this section we introduce some concepts and definitions that are used throughout the paper.

A *workflow* is represented by a Directed Graph (DG) W = {Nodes, Links}, where: 1) Nodes is the set of *workflow activities*, and 2) Links is the set of *dependencies* between workflow activities.

W is a static element that specifies the structure and possible orchestration of workflow activities and is commonly specified by a workflow language. The workflow language is usually represented in a textual manner, although graphical interfaces have been employed for facilitating the interaction with the workflow system. The size and complexity of workflows varies, and while a graphical representation may be optimal for simple workflows, this type of representation is not feasible when scaling the number of workflow activities. In such situations, a textual representation is better suited. Furthermore, even when graphical interfaces are employed the graphical language is associated with a textual representation for storing and managing workflow instances in files [30]. In many cases XML is used for the textual representation but scripting languages can also be employed.

Ports serve as containers of data associated with workflow activities. Ports also state communication channels between activities providing entry (input ports) and exit (output ports) points to the workflow activities. For Grid workflows it is common to assume that ports have no associated type information and that workflow activities internally distinguish the correct semantics of the data in ports. In Section 5.1 we consider the implications for the cases where the ports are typed (typical in local workflows). Communication to and from nodes is performed by specifying links between ports associated to different activities. The links represent dependencies whose nature, i.e., control or data flow, is unimportant from a representation point of view.

A *workflow activity* represents a unit of execution in a workflow. An activity can be an indivisible entity or it can be a sub-workflow containing other activities. Associated input ports provide the required input for the activity while the output is produced through output ports. As such, activities can be treated as functions whose domain is given by the cross product between input ports and whose range is given by the cross product of the output ports. A distinctiveness of Grid workflows is that workflow activities are stand-alone command line applications that require a mapping between the expected command-line arguments and the input and output ports used for communicating with other activities. There are different manners in which this mapping can be achieved, e.g., see [8] and [69]. For the rest of the paper we assume that this mapping has been performed so that workflow activities are enacted with the appropriate parameters and the required information is moved to and from activity ports.

A *Model of Computation (MoC)* is an abstraction that defines the semantics in which the execution of a workflow W is to be carried out. A *workflow engine* is a software module that given a workflow description (W) and a MoC (M) as input, executes the activities in W following the rules specified by M. Thus, a workflow engine provides a concrete implementation of a MoC and it is responsible for enacting workflows. The enactment is performed by selecting activities to execute. The manner in which the activities are selected and how the communication between activities is carried out is defined in the MoC. The engine can execute in a local machine or it can be exposed as a permanently available service accessible by many users. The latter useful when executing long processes, as tools can reconnect to the engine for monitoring and managing purposes without requiring permanent connections.

4 Model of Computation (MoC)

A model of computation is a formal abstraction that provides rules to govern the execution, in this case, of workflows. Programming and workflow languages have traditionally been designed with an underlying abstraction whose formal semantics is given by a MoC [35]. Similarly, workflow engines instantiate a MoC when enacting workflows. While it is common for workflow systems not to reveal the MoC used by the engine, there are some systems in which the explicit selection of MoC is required[1].

Different MoCs have different strengths and weaknesses. Selecting a MoC often depends on, among other things, how well the model abstracts the problem at hand and how efficient the model can be implemented. A too abstract specification, for the model, is not only inefficient but is also unfeasible to implement while too much detail in the specification over-constraints the model making it inefficient and more costly to implement [35]. In essence, for a MoC to be efficacious there should be a balance between the generality offered by an abstract specification and the particularities of a detailed one. Such a MoC is useful not only for a range of scenarios but it is also possible to model and analyze interesting aspects of individual models.

Several MoCs have been used for the general workflow case, e.g., Petri nets, dataflow process networks, and UML activity diagrams. Some of these MoCs are not suitable for scientific workflows, e.g., even though BPEL is widely used for workflows in a business context, it is not as popular in the scientific community. This, despite several attempts at adapting BPEL for

[1] e.g., in Kepler [43] MoCs are exchangeable and are called *Directors*.

the scientific workflow peculiarities [22, 42]. Instead, dataflow approaches have predominantly been used for scientific workflows [43, 51, 64]. According to McPhillips et al. [48] this adoption is due to the inherent focus on data in the dataflow MoC, a characteristic that resembles the scientific process.

Still, a straight forward[2] adoption of dataflow for Grid workflows is not suitable. This is due to the characteristics of current, and in the foreseeable future, Grid execution environments, such as the typical lack of control over the internal states of workflow activities and the impossibility of continuously streaming tokens between activities. As presented in Section 3, a distinctiveness of Grid workflows is that they typically consist of a number of independent command line applications that are configured by environment variables or command line arguments. In this setting, activities are considered black boxes and it is impossible for the workflow MoC to control their internal states. For example, to the Grid workflow MoC, activities are considered to be executing once they are scheduled for execution on a Grid resource, even though they in practice may be stalled in a batch queue. Another distinctive characteristic of Grid workflow MoCs is that, as opposed to the continuous streaming of tokens found in dataflow networks (e.g., as in [43]), activities execute only once and communicate with other activities at the end of this execution. Thus, it is important for the Grid workflow MoC to support asynchronous communication and to carry out all communication only when activities finish executing, disabling those activities that finish executing.

Because of the previous restrictions, MoCs that have been typically used for Grid workflows are limited to Petri nets or some type of control flow MoC specified either by DAGs [11] or by specialized imperative languages [61, 67]. Below we present the manner in which Petri nets have been used as a MoC for Grid workflows. We also make observations on the reasons why a dataflow approach is not commonly employed in Grid workflows.

4.1 Petri nets

Petri nets are a mathematical and graphical tool specially useful for modeling and analyzing systems that are concurrent, asynchronous, and nondeterministic. Based on Murata [50], a Petri net is a bipartite graph in which nodes called *places* are linked to nodes called *transitions* by directed edges. There are no elements of the same node type connected to each other, e.g., places are only connected to transitions and not to other places. Places

[2]By straight forward we mean using the same MoC , without changes, as in local workflow systems.

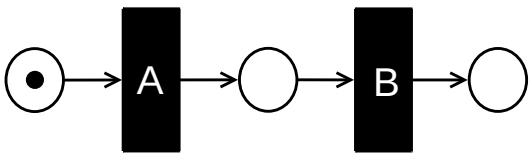


Figure 1: Petri net illustrating the control flow between activities A and B.

directed to transitions are called input places while places coming out of transitions are called output places. Places contain tokens that are required for enabling (initiation) the firing of transitions. Places also have an associated capacity that indicates the maximum amount of tokens they can hold. Edges have an associated weight that indicates how many tokens are required to enable a transition as well as how many tokes are produced when a transition is fired. A transition is fired only when each input place has the necessary tokens, specified by the edge weight, to enable that transition and if the output places have not yet reached full capacity. Once fired, an amount equal to the edge weight is set on each output place. The *marking* of the net describes its state and is given by the distribution of tokens in the places. The *initial marking* describes the state of the net before any transition has fired and a new marking is created after each firing of the net.

Petri nets have traditionally been used for representing the control flow of workflows [30, 33, 60, 65]. The manner in which this flow is represented is illustrated in Figure 1. In this network, two activities, A and B, are executed in sequence. Activity A is enabled (i.e., ready to fire) as there is a token in its input place (represented by the black dot). Conceptually, the firing of A symbolizes the execution of some activity in a Grid resource. When A completes execution a token is placed in the output place of A, which in this case is also the input place of B, thus enabling B. It is important to notice that B is not able to execute until A has finished. For this net the tokens not only symbolize the passing of control between activities, but they also maintain the state of the net. There is however no explicit information about the data created or consumed by the activities.

The limited support for combining control and data flow within the same model has been addressed by the introduction of specialized high-level nets, in particular *Coloured Petri nets (CPN)* [36]. In CPNs places have an associated data type (color set) and hold tokens that contain data values (colors) of that type. Arc expressions indicate the number and type of tokens that can flow through an arc. Tokens of the specific data types

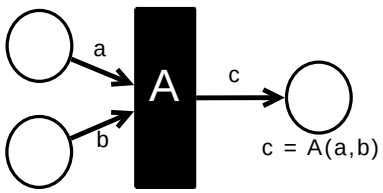


Figure 2: Representation of workflow activities using Coloured Petri nets. Based on the work in[34].

need to be present in its input places for a transition to fire. Transitions can also have an associated *guard*, a boolean expression that enables the firing of the transition whenever the guard evaluates to true. There is no ordering in how the tokens are consumed by a transition with respect to how they arrived to an input place. A queue can be associated with a place if ordering is desired. A more detailed discussion about this type of networks can be found in [36, 50]. Jensen [36] presents a more informal and practical introduction to CPNs whereas Murata [50] briefly touches upon the relationship between High-level nets, a group to which coloured nets belong to, and logic programs.

In the Grid workflow context Petri nets and CPNs have been used both as a graphical specification languages and as a workflow engine MoCs. Guan et al. [30] employ simple Petri nets as graphical language for defining workflows in the Grid-Flow system. Workflows defined with Petri nets are translated to the Grid-Flow Description Language (GFDL). Workflows in GFDL are then fed to the Grid-Flow engine. Language constructs[3] such as *OR-Split*, *AND-Split*, and *AND-Join* are used to generate instances of choice, loops, and parallel structures offered by GFDL. Hoheisel and Alt [34] employ CPNs both as specification language and as a MoC. In the latter case, transitions are used as processing elements (i.e., workflow activities) in which data tokens are distinguishable. Thus, transitions operate as functions whose parameters are obtained from the input places and the results are stored in the output places.

Figure 2 illustrates this process, where the result of applying the function in A to the parameters a and b is stored in the output place c. This Petri net models the data flow generated by the data files produced and consumed by the transition (representing a workflow activity) A.

While there are many characteristics that make Petri nets a sound choice

[3]In some settings called Workflow Patterns [66].

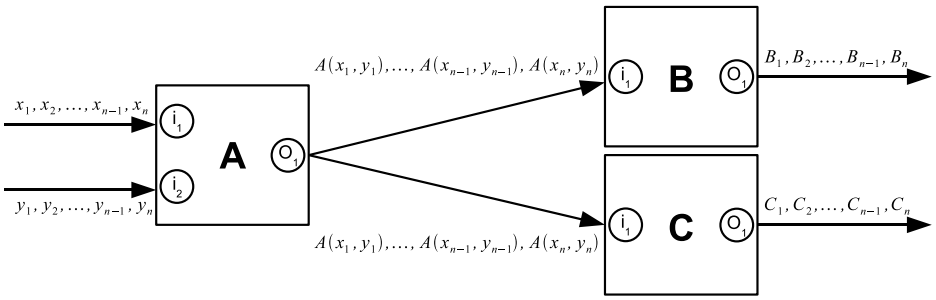


Figure 3: A dataflow network that instantiates concurrent execution of activities B and C.

for a Grid workflow MoC, there are some issues to be resolved. For example, Murata [50] identifies that Petri nets can become quite complex even for modest-size systems. A weakness of Petri nets when compared to a dataflow approach is the necessity to define parallelism explicitly e.g., using *AND-Split* and *AND-Join* [66].

4.2 Using dataflow networks on Grid workflows

Dataflow networks are the preferred MoC for local scientific workflows. For example, Triana [9], Kepler [43], and Taverna [52] offer capabilities that, one way or another, resemble the dataflow style of computation. In the original dataflow approach the focus was on fine-grained flows at the CPU instruction level. In those cases nodes represent instructions and edges represent data paths. When this metaphor is moved to the workflow paradigm, nodes no longer represent instructions but coarse-grained processing operations while edges represent dependencies between workflow activities.

Figure 3 illustrates a dataflow network in which activity A sends tokens concurrently to activities B and C. The figure presents a simplification of the actual process been carried out, nevertheless it helps us present the problems found when attempting to apply a dataflow approach to Grid workflows. The Figure shows a pipeline dataflow in which initial tokens are processed by A, and then B and C concurrently process the tokens generated by A. In the original dataflow process networks (e.g., as presented in [59]) tokens are continually streamed through the pipeline so that activities A, B, and C are all concurrently processing although operating on different tokens. The

circles inside the rectangles represent ports that serve as containers of data (see Section 3) and also serve as interfaces for establishing communication channels between workflow activities. In local workflows these ports have an associated data type that indicates the type of tokens that they can hold. These data types need not be restricted to simple types (e.g., integer, float, or string) as they can also be complex data structures [48]. On the contrary, in Grid workflows tokens only represent associations with data files and are otherwise untyped. Further discussions about type systems in workflows is presented in Section 5.1.

We identify the continuous streaming of tokens between activities and the lack of control of the workflow MoC over the internals of the activities as the main impediments for adopting dataflow style of computation in Grid workflows. Below we present a brief discussion on how these issues can be addressed.

Streaming of tokens between activities. In dataflow nets parallelism is achieved through concurrent processing of tokens by different activities. This e.g., can be seen in Figure 3 when A is processing token (x_i, y_i) while B and C are processing tokens produced by $A(x_{i-1}, y_{i-1})$, a previous execution of A.

In local workflows, this process is easily accomplished by e.g., interprocess communication or message passing. The nature of the Grid, however, impedes an easy solution if attempting to implement the same functionality on Grid resources. For Grid workflows, resources where activities A, B, and C are to be run must be *guaranteed* to start executing at the same time, a process known as *co-allocation* [21]. All resources must also be able to synchronize with each other to establish direct lines of communication between themselves.

The process of co-allocation of Grid resources is difficult for multiple reasons. At a technical level activity A must have access to the network addresses of the resources where B and C are running. However, this information is often not distributed outside the site in which B and C are running, making such a synchronization impossible. Another technical issue is that since many Grid applications operate on very large data files, transmitting only small bursts of data is not efficient. Furthermore, streaming tokens between activities requires adaptation of applications that normally communicate through files.

Several solutions for the problem of co-allocation of Grid resources have been proposed. These solutions usually depend on advance reservations

to ensure that all resources are available at the same time. However, the use of advance reservations introduces a problem at a managerial level, as reservations are known to degrade the performance of a system [45, 62, 63].

Lacking globally controlling coordinators. In local workflow systems the MoC has control over the internal processing of the activities. This means that any changes in the internal states of the activities are exposed to the MoC. For example, for the case illustrated in Figure 3, the MoC can recognize the state that activity A reaches after processing token (x_i, y_i).

This is not the case for Grid workflows. In this setting the MoC can only recognize that an activity is ready to execute, that an activity has been submitted to a Grid resource but for practical reasons can be considered to be executing, and that an activity has finished executing either successfully or with an error. All other changes in state are transparent to the MoC.

The nature of this obstacle is the use of command line applications that operate on *un-structured*[4] data files. However, the use of command line applications also simplifies the use of Grid resources by end-users as they are not required to modify their software. Thus, there is a trade-off between having simple coordinating MoCs in which applications can be easily included, and having more complex MoCs that require modifications to applications (even complete re-implementations) prior inclusion in the model.

While Petri nets have previously been used to model dataflow [40, 68], the use of CPNs facilitates this process. The CPN in Figure 2 can be adapted to model a processing unit from a dataflow network, i.e., the input and output places have similar functionality as input and output ports. A difficulty when modeling dataflow with CPNs is how to describe the implicit parallelism found in dataflow networks. A naïve approach produces conflicts among the concurrent activities. This can be seen in the CPN in Figure 4 that attempts to model the dataflow network of Figure 3. The conflict occurs after A fires and sets a single token in its output place. At this point, both B and C are enabled but only one can fire as there is only one token to consume.

Figures 5 and 6 illustrate two manners in which this problem can be resolved. The first approach is to add one output place for each transition that depends on the output place whereas the second approach adds a subnet. With the second approach there is a complete separation between the activities as opposed to the first approach in which the output and input

[4] As opposed to structured data files such as XML documents defined through XML schema or basic data types such as integer or string.

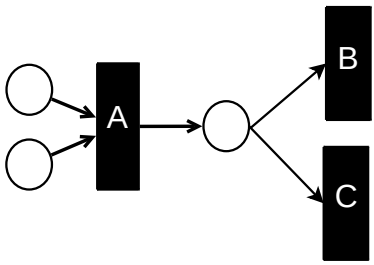


Figure 4: CPN that simulates the functionality of Figure 3. In this CPN activities B and C are in conflict.

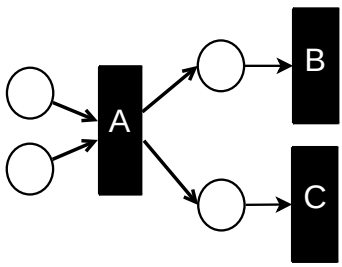


Figure 5: A CPN that requires one output place for each dependent activity. In this case one for each B and C.

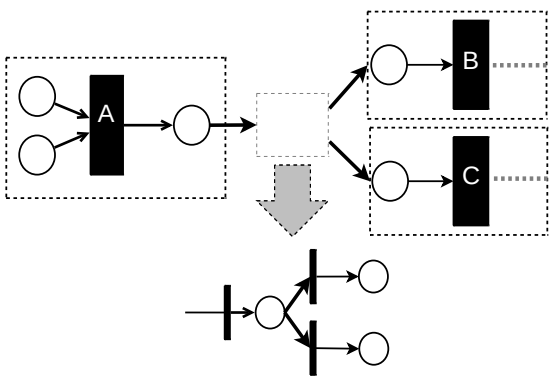


Figure 6: A CPN that uses a sub-net to connect dependent activities. The output place of A is not the same as the input places of B and C, thus each can have an associated buffer.

places are shared between the activities. This second approach provides a more accurate representation of dataflow processing units.

A MoC inspired by dataflow networks but adapted to Grid environments is presented in our previous work [20]. In this approach, activities execute once and are then disabled from further execution. Input and output ports are used to establish dependencies between activities. The only data type used is string and data values represent Grid storage locations where the data files are to be found. Tokens carry this information from one activity to the other. Files are transferred to the resources where the activities are to execute from the locations where previous activities stored their outputs.

While this discussion has focused on Petri nets and dataflow, it is important to mention that by far the most common approach for controlling dependencies in Grid workflows is through DAGs. Currently, there are several projects [11, 15, 38] that offer higher level interfaces for specifying workflows that are rewritten in order to reduce execution time. The output of these rewrites is often produced as DAGs. In these cases the DAGs represent schedules for execution of workflow activities on Grid resources.

5 Workflow Representation

In this section we address several factors relevant to workflow languages. We begin by exploring the use of *type systems* in workflow languages (Section 5.1). We explore the differences in capabilities offered by type systems depending on whether the language is for local or Grid workflows. As in the MoC case, the nature of these differences arises from the differences present in Grid environments.

A common differentiation in workflow languages is that between control-driven (control flow) and data-driven (data flow) styles for representing workflows. In control-driven workflows the complete order of execution is specified explicitly whereas in data-driven workflows only the data dependencies between activities are given and the execution order of the activities is inferred from the manner in which the data dependencies are satisfied. Thus the data-driven style specifies a partial order for the workflow activities and the exact order of execution is not known until run time. The choice of style is mostly driven by the selected MoC. However, several languages support mix-models, based on one MoC but with basic support for the other. In Section 5.2 we illustrate mechanisms to translate control-driven workflows into data-driven ones and vice-versa.

All modern general-purpose programming languages are Turing com-

plete, with the caveat that their run time environments as provided by today's computer hardware have a finite memory size as opposed to the infinite tape length of the universal Turing machine. Being Turing complete implies that a system is computationally equivalent to other Turing complete systems. Within the workflow community there are arguments for and against having Turing complete workflow representations. For example, according to Deelman [13], "one have to be careful not to take workflow languages to the extreme and turn them into full-featured programming or scripting languages". On the other hand Bahsi et al. [5] argue that workflows without conditions and iterations are not sufficient for describing complex scientific applications. Nevertheless, when examining current workflow systems [5, 14, 56] it becomes apparent that most systems are Turing complete[5]. In light of this finding, we discuss in Section 5.3 the manner in which the prerequisites for Turing completeness, namely *state management*, *conditions*, and *iterations* are implemented by different workflow languages.

5.1 Workflow languages and type systems

One aspect to consider about workflow languages is the choice of type system offered by the language. Which data types are present, whether data types must be explicitly specified or if implicit specification is supported, and whether the language must provide mechanisms for describing new types are all issues that vary among languages, specially between languages for local and Grid workflows.

For local workflows, it is relatively easy to wrap applications in an embedding model that ensures compliance with the type system. For this case activities are often developed from scratch, using APIs provided by the workflow system and are thus completely capable for operating within the framework provided by the workflow system. This is the case in e.g., Triana [9] and Kepler [43]. However, due to the use of command line applications, applying this functionality to Grid workflows is not simple. Data type information is not required for command line applications and thus it is not included in any one of the several job description languages (e.g., JSDL [2]) currently in use. This is the case also for the languages in e.g., DAGMan [11] and GWEE [20]. Nevertheless, there are some Grid languages that use different methods for including type information in the workflows, this is the case in e.g., BPEL [37], ICENI [47], and Karajan [67].

[5] As a side note, non-Turing complete languages have many uses [7], although they are typically used in specialized areas.

Exposing Grid applications through type interfaces, e.g., Web services or component models such as CCA [3] or GCM [10], can be a substantial effort as it requires e.g., software installations on remote machines [16, 46]. A further complicating factor when adding a type system to Grid workflow languages is that such languages most often lack support for defining custom data types. It is thus hard to express the structure of data files that are used by a given application. An exception is the Virtual Data Language (VDL) [69] that provides primitives for describing data sets, associating data sets with physical representations, and defining operations over the data sets. Furthermore, whereas it is relatively simple to verify that input values adhere to certain basic types (e.g., integer or float) it is more complex to verify that a data file is of a specific format or follows a predetermined structure.

In summary, using type information in Grid systems simplifies workflow design and error handling, but it also adds overhead as each application that is used must be exposed through a type interface. On the other hand, not using a type system (e.g., supporting a single data type) increases flexibility as any (command line) application can be embedded in a workflow, but this at the expense of higher difficulty in detecting data incompatibilities.

5.2 Control-driven and data-driven workflows

Control-driven and data-driven MoCs differ in the semantics of consuming dependencies between activities. In control-driven workflows, consuming a dependency results in the transfer of execution control from the preceding activity to the succeeding one, whereas in data-driven a data token is sent from the first activity to the next one and an activity is only able to execute after all data dependencies are cleared (i.e., all tokens are received). The choice between styles depends, in part, on the applications to be described, e.g., some areas such as image or signal processing have traditionally been represented with data-driven workflows as this model provides a natural representation of the problems studied within these fields [54, 64].

Recent results advocate for a simple hybrid model based on a data-driven style but extended with limited control constructs [14]. The validity of this model is attested by support of both styles of flows by several contemporary systems [15, 38, 43, 55, 61]. In part, such a hybrid model can be realized because, despite their differences in operation, converting from one MoC to the other is not a complicated process. However, it is possible that performing such conversions, control to data and data to control, requires simulating missing functionality with the primitives available in either style.

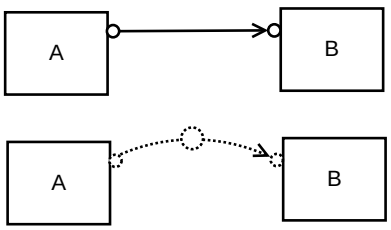


Figure 7: A control-driven dependency (top) denoted by solid lines and the corresponding data-dependency (bottom) illustrated by dashed lines. In both cases activity B executes after activity A.

This results in more complex workflows and increases the risk of introducing errors. Nevertheless, control- and data-driven workflows are interoperable at the workflow language level. Below we present a manner in which such conversions can be attained for the case of Grid workflows.

5.2.1 Conversions between control-driven and data-driven workflows

As presented in Section 3, communication between activities in Grid workflows is performed by the transfer of untyped data files. For this case control dependencies can be converted to data dependencies by using tokens that carry no data values (i.e., *dummy data tokens*) and in practice only represent the transfer of control from one activity to another. This case is illustrated in Figure 7 where equivalent control- and data-driven versions of a control dependency between activities A and B are presented. The circle in the bottom workflow being the dummy data token introduced to simulate the transfer of control from A to B. The case with actual data tokens is presented in Figure 8. In this case a file transfer (Activity B) that is represented explicitly in the top workflow is converted to an implicit transfer embedded in a data dependency as shown in the bottom workflow.

The reverse process can be applied for converting from data- to control-driven workflows. Data dependencies are converted by inserting an intermediate activity that performs an explicit file transfer from the location where the source activity was executed to the machine where the target activity is to be executed. The top workflow in Figure 8 illustrates the result of this process, converting from the bottom workflow in the figure. Notably, it is possible to eliminate the explicit file transfer (B) if such transfer is per-

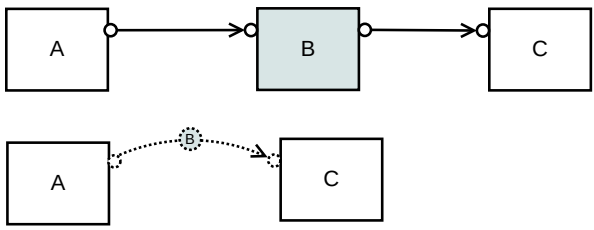


Figure 8: File transfers are workflow activities in control-driven workflows (Activity B in the top workflow) and data dependencies in data-driven workflows (data dependency with token labeled B in the bottom workflow).

formed as part of the job execution, a mechanism supported by most Grid middlewares (e.g., Globus [25]). For abstract workflows[6], as the resources where activities are executed are not know until enactment, the conversion must be performed after executing the source activity as the actual location of the data files is not known before. Otherwise the workflow must specify the resources where all activities are to be executed (i.e., it must be a concrete workflow).

An important step when converting from data- to control-driven is to perform a *topological sorting* of the data dependencies. Consequently, the resultant control-driven workflow specifies only one among many possible execution orders. As a result the precise execution order may differ between the data and the derived control-driven versions. This is however of no practical concern, as the respective execution order of all dependent activities is maintained.

In practice these conversions have been performed a number of times. For example, Mair et al. [44] describe how to convert both styles of workflows, control- and data-driven, to an intermediate representation based on DAGs. A concrete implementation between the Karajan language (control-driven) and the internal representation of GWEE (data-driven) is presented in our earlier work [20].

5.3 Essential language constructs

Turing completeness, the ability to simulate a universal Turing machine, is important when analyzing the computing capabilities of systems. The ability

[6]In abstract workflows only the structure of the workflow is specified. The physical resources where the activities are to be executed are not specified.

to manage state, e.g., by been able to define, update, and read variables, is one criteria for Turing completeness. In addition to *state handling*, *condition* (branching) and *repetition* (typically recursion or iteration) functionalities are also required mechanisms for Turing completeness. Here, we look at workflow languages in light of these mechanisms. Motivating use cases are presented for each mechanism as well as the manner in which the mechanisms are implemented by different workflow languages. We also take a look at when and how *Collections* are useful.

5.3.1 Workflow state management

In modern imperative and object-oriented programming languages state is managed by defining and updating variables that represent an abstraction of memory locations. Some workflow languages, such as Karajan [67] and BPEL [37], support state management through a variable construct similar to that of modern programming languages. Other workflow systems, e.g., DAGMan [11] and Taverna [52], have no built-in language mechanism to manage state. The only state in those systems is the run time state of the workflow activities (e.g., completed, running, waiting). Not having variables creates difficulties when using general condition and iteration constructs as these make branching decisions mostly based on state. A different approach for implementing a state-like mechanism is to use system parameters for defining properties that hold similar functionality as environment variables. This mechanism is used by ASKALON [61] and JOpera [55].

5.3.2 Conditions

By far the most common use for conditions in workflows is for *workflow steering*, a functionality that carries similar semantics as the well known *if* and *switch* constructs. The idea is that the flow is *dynamic* and the output is *non-deterministic*. The most frequent use case for workflow steering is classical flow of control where the branch to enact is decided at run time based on the outcome of previously executed activities. Changing the enactment of a workflow in reaction to external events is an alternative steering use case suggested by Gil et al. [27].

Another use case for conditions is *iterative refinement* scenarios, where some activity needs to be repeatedly executed until a condition is met. In addition to conditions, iterative refinement scenarios requires iteration constructs and a testing mechanism, both issues discussed in more detail later in this section.

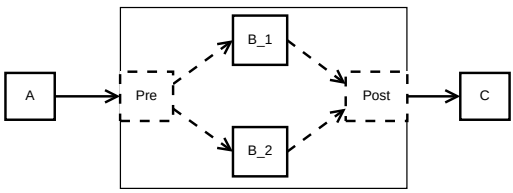


Figure 9: The black box approach where conditions are hidden in a special activity type. In this case, the condition is transparent to the rest of the workflow.

A final use case in which conditions have been employed is *fault tolerance*. A survey of fault tolerance mechanisms used in various Grid workflow systems is found in Plankensteiner et al. [57]. In this use case conditions are used for defining alternative actions for situations in which a workflow activity fails. In contrast with programming languages that typically have special language constructs for catching generated runtime exceptions, in distributed Grid workflow environments, errors in remotely executing activities do not generate such exceptions but rather result in *failed activities*. Conditional statements are typically sufficient for many cases of fault tolerance.

There are basically two abstractions for implementing conditions in a DAG or dataflow based workflow representation. The first one considers conditions as a special type of workflow activity. In this approach, illustrated in Figure 9, the condition is viewed as a black box with the branches hidden inside the activity. This form of condition gives rise to so-called "*structured workflows*" [41] which are analogous to conditions and iterations found in structured programming. This type of constructs have only one entry (pre) and one exit (post) point into and out of the workflow. An example of this approach is Triana [9]. The other alternative is to have the condition as an activity that selects a branch of execution but with all possible branches exposed in the main workflow. In this case, care must be taken as deadlock may arise if the branches are not well synchronized. This second approach is how conditions are implemented in JOpera [55] and Karajan [67].

After a branch is selected, data must be sent to the initial activity of the branch in order to trigger execution. However, unless special care is taken, not-selected branches may end up in a dead-lock state, waiting for input forever. One solution to this undesired effect is to prune the workflow graph by removing not-selected subgraphs from the workflow. Such a solution is

akin to the elimination of dead code, a well-studied problem in compiler theory [1]. The introduction of conditions can also introduce problems with synchronization with previous branches of the workflow. More specifically, combination of primitives such as OR-Split and AND-Join in BPEL [37] may result in dead-locks unless care is taken.

Detecting workflow termination becomes more complicated when the workflow contains branches that do not execute. With conditions implemented using the black-box approach, this problem can be solved by marking conditional activities as completed once one of its branches finish executing. A different approach is to mark branches along the non-selected paths with a terminal state that indicates that they are not to run. It is also possible to tag certain activities (such as activity C in Figure 9) as terminal ones. Once such an activity completes, the workflow enactment engine is assured that the workflow has finished executing.

In a typical, non-typed Grid workflow, condition evaluation (often referred to as testing) is hard to achieve as the workflow system has no control over the evaluation of the condition. Generality in the testing capabilities is also difficult to achieve unless the system limits what can be tested. As activities in Grid workflows usually communicate via files instead of typed variables, ordinary boolean testing is tedious in a Grid environment. To complicate things further, it is typically hard to distinguish between errors in the application execution and faults related to the Grid infrastructure. One possible solution is to offer a subset of predefined testing capabilities,as it is done in e.g., UNICORE [5, 17]. In this work, three sets of tests are defined: (1) ReturnCodeTest, indicating successful or failed task execution; (2) FileTest, for checking whether files exist, are readable, writable, etc.; and (3) TimeTest, that tests if a certain time has elapsed. These evaluation capabilities are based on information similar to what is known about process execution in shell scripting languages. An alternative method is to implement testing by an external agent that has the domain-specific knowledge required to perform comparisons [14].

5.3.3 Iterations

We distinguish between three types of iterations:

1. *Counting loops without dependencies* between iterations are often referred to as parameter sweeps or horizontal parallel iterations. This type of iteration generates parallel independent branches and is akin to applying a function to each element from a set. In many program-

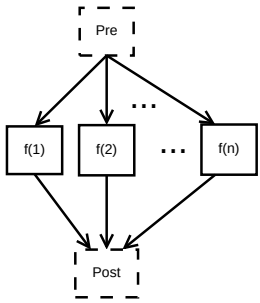


Figure 10: A counting loop without dependencies expressed as a parameter sweep.

ming and workflow languages, such loops are often expressed using language constructs such as *parallel-for* or *for-each*.

2. *Counting loops with dependencies* between iterations where the results from one iteration is used in the next one. This type of loop can hence not be independently executed in parallel. Typical syntax for these iterations is *do-n* and *for-n*.

3. *Conditional loops* are also referred to as non-counting iterations, temporally dependent iterations, or sequential iterations. This type of loop stops only when a certain condition is met. *While*, and *do-while* are used to express conditional loops in most programming languages.

Algorithm 1 Counting loop without dependencies

```
1: for I ← 1 ... N do
2:     f(I);
```

As demonstrated by Ludäscher et al. [43] counting loops without dependencies can be expressed using the *map* function from functional programming, that is, $f(x_1, x_2, \ldots, x_n) \Rightarrow (f(x_1), f(x_2), \ldots, f(x_n))$. Algorithm 1 illustrates a typical loop of this type and Figure 10 illustrates the equivalent workflow construct after applying the map function. This type of loop give rise to a high degree of concurrency as the threads of execution are completely independent. The same concurrency is impossible to achieve for counting loops with dependencies. The reason is that an iteration depends on results from a previous iteration. However, as illustrated in Algorithm 2

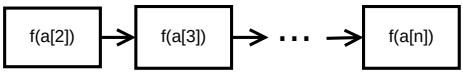


Figure 11: A counting loop with dependencies between iterations rolled out.

and Figure 11, this type of iteration can be rolled out [1, 12] and equivalent functionality can be provided without using iterations. Contrary to the two types of counting loops, conditional loops cannot be expressed by rewriting the workflow graph. Furthermore, this type of loop requires support for conditions to test when the terminal condition is met. We remark that it is trivial to rewrite a counting iteration as a conditional loop, whereas the opposite is not possible in the general case.

The mechanisms to support iteration constructs by workflow languages are similar to the ones used for conditions. In the first case, the black box approach, the iteration is a special workflow activity with one entry and one exit point. This is the approach taken by the extensible actor construct in Kepler [43]. The second alternative is to have an expression-like construct that allows the flow of control to iterate over selected activities in the workflow. This second approach introduces cycles to the workflow graph and creates a more complex enactment since care must be taken to avoid infinite loops.

Iterations, essentially being a flow of control construct, are easy to support by control-driven languages, whereas the semantics of iterations are unclear for pure data-driven languages. Mosconi et al. [49] investigates the minimal set of control flow constructs required to support iterations in a pure dataflow language and surveys existing implementations of iterations for visual programming languages.

Algorithm 2 Counting loop with dependencies

1: $a[1] \leftarrow initial\_value$;
2: **for** I $\leftarrow 2 \ldots N$ **do**
3: $\quad a[I] \leftarrow f(a[I-1])$;

5.3.4 Collections

Some programming languages, e.g., LISP and Perl, have built-in *for-each* operations that treat a collection of elements as a single entity. Similar ideas are used in vectorizing and parallelizing compilers [4, 32]. These type

of data-collection operations are also supported by some workflow systems, e.g., ASKALON [58] and the COMAD [48] implementation in Kepler. Data-collection mechanisms are data-centric and hence can simplify the use of dataflow style languages. However, collections do not bring additional functionality beyond what is offered by parameter sweeps or iteration constructs except the aforementioned simplification.

6 Discussion and Concluding Remarks

Here we discuss the topics covered in the previous sections with a comprehensive outlook. As such, while topics are ordered as they are introduced in those sections (Sections 2–5), there are cases in which the topics overlap section crossings.

6.1 Model of computation

The model of computation is the central concept of a workflow engine. It can even be said that the workflow engine is merely an implementation of a MoC. Previous results [13, 48] suggest that a dataflow approach suits the scientific process best. This is supported by the number of solutions that use the dataflow MoC, and the manner that these solutions can be trivially adapted to operate in disparate scientific fields. Common to these solutions is the use of local machines as execution environment, which appears natural as the environment that local machines offer is well adapted for the dataflow MoC.

This is not the case for Grid workflows. Limitations such as the lack of control over activities, lack of support for streaming tokens between activities, and the unavoidable requirement of executing activities in a batch processing fashion, make the use of a pure dataflow MoC unfeasible to achieve. Control-driven approaches appear to be better adapted for this type of environment. Nevertheless, there are several projects that attempt to use dataflow style of coordination for Grid workflows.

One way to achieve functionality similar to what the dataflow MoC offers is to use higher level representations that are later refined to concrete activity specifications. This is the case presented in, e.g., [8], [15], and [26]. In [15] and [26] a concrete workflow DAG with the correct order of execution for the activities is generated from more abstract representations, whereas in [8] a data-driven representation is concretized into dataflow-like workflows that are enacted on the Grid [20]. As presented in Section 4.2, CPNs can

also be used for representing dataflow style coordination but care must be taken to avoid (firing) conflicts.

Interoperability between MoCs can be achieved under certain circumstances. The manner in which the workflow activities are implemented, in particular the MoC underlying their design, is the key aspect that enables the activity to operate under different MoCs. On the local side Goderis et al. [29] provide insight on which combinations of MoCs are valid and useful. This work offers a hierarchical approach in which MoCs, called *directors*, require certain properties from the activities, called *actors*, that they coordinate. Directors also export properties to the actors in which they are included. The set of properties offered and required establishes a contract and depending on how well the contract is respected it assures the compatibility of actors and directors, and thus the potential compatibility among different MoCs. Actors that completely adhere to the contract are called *domain polymorphic* [18] and can be used by any director. Thus, when seeking interoperability, it is important to develop the workflow activities in a manner in which they can be controlled by different MoCs. However, this is not always possible.

The core of the difference between local and Grid workflows is the execution environment. Interoperability between local and Grid MoCs is thus possible only in a few cases and directly depends on the manner in which the activities are executed on the Grid. Nevertheless, in practice this type of interoperability is not often requested, instead, what is commonly expected is for local workflow systems to be able to submit work to the Grid on either a per activity or per sub-workflow basis. The latter case with aid from a Grid workflow system.

6.2 Language issues

Section 5.1 discusses the issue of type systems and workflow languages. As presented there, it is common for local workflows to support type information while it is much less common for Grid workflows to support this functionality. In the reminder of this section we revisit the various language constructs introduced in Section 5 and investigate the extent to which the respective mechanisms are required. For the various constructs, we look at typical use cases and discuss whether equivalent functionality can be achieved by different means or if the use case motivates the particular language construct used.

6.2.1 State management

Variables have traditionally been used to manage state in programming languages. This is also the case for the workflow language of Karajan [67]. However, lack of a variable construct need not imply that a language is not Turing complete. For example, Glatard et al. demonstrate how to implement a Turing machine in the Taverna Scufl language [28]. In this work, the limitation of not being able to define variables (and hence manage state) in Scufl is circumvented by performing state management inside one of the workflow activities (implemented in Java). For Petri nets it is also possible to handle state. For this case the state is given by the marking of tokens in the places of the net.

6.2.2 Conditions and iterations

It appears that in order to describe and execute anything but the most trivial process, workflow steering, and hence conditions, are required. However, the survey by Bahsi et al. [5] shows that not all workflow systems support conditions as part of their workflow language. Equivalent functionality can be achieved by other mechanisms, as illustrated e.g., by the pre and post scripts that are used to steer the path of execution in DAGMan [11]. This suggests that although conditions are required for workflow steering they need not necessarily be part of the workflow language as they can be expressed using alternative mechanisms. Instead, at least for Grid workflows, a mechanism to implement the testing required for conditions is more important.

The iterative refinement use case can be implemented in two ways. One alternative is a fine-grained workflow that iterates over individual activities until some condition is satisfied. In addition to handling conditions and testing, this approach requires the workflow language to expose a loop mechanism. Alternatively, it is possible to have a coarse-grained workflow in which the activities as well as the testing mechanism are all abstracted and hidden inside a single workflow activity. In this latter approach, which is taken e.g., by Kepler [43], conditions are not necessary for specifying the workflow. The support for the iterative refinement use case is hence a trade-off between (potentially too large) granularity, and thus possible limited parallelism, and added complexity of the workflow language.

There are two types of failures occurring frequently in workflows systems. In the first type, infrastructure problems such as network failures, power outages, temporarily unavailable storages or databases, insufficient disk space, incorrect hardware, etc. cause an activity or a file transfer to

fail. For this case a lower level tool would ideally ensures fault tolerance, e.g., by restarting interrupted file transfers, resubmitting failed jobs to alternative machines, etc. These types of mechanisms to recover from infrastructure problems are known as *implicit fault management* [33]. Such a recovering infrastructure removes the need for the user to manually, through conditions, encode alternative execution paths for the workflow to follow upon failures. In contrast the manual alternative quickly becomes unfeasible due to the large number of potential error sources. In the second type of failure the workflow enactment fails due to errors in the workflow itself. These errors can be faulty descriptions of activities, incompatible messages exchanged between activities, unintended deadlocks in the workflow graph (e.g., circular data dependencies), etc. For this case conditions are of limited use as the errors in the workflow are detected only during enactment whereas conditions must be added at design time. Manual inspection and modification of the workflow is typically required to solve this type of problems.

Similar to the case of conditions, a repetition mechanism is often required to express complex workflows. The mechanism need not be an iteration construct in the language, as it is commonly known that recursion offers the same functionality. The latter approach is taken by e.g., Condor DAGMan [11] and JOpera [55]. Another example of a repetition with no explicit construct is to use parameter sweeps for implementing loops without dependencies. For this case, a mechanism for distributing data, e.g., data collections, to the different threads of execution (each operating on a different iteration from the loop) often facilitates this process. As far as an iteration mechanism is required, there is always the possibility of using a single application that hides the iterative structure of the workflow. However, care must be taken in order not to limit potential concurrency and thus reduce the performance of the workflow.

6.3 Execution environment interoperability

It is difficult to address the issue of execution interoperability for local workflows as workflow activities are developed specifically for a particular system. These activities are dependent on libraries that offer a common execution and communication environment for operating within a particular system. It is not easy to decouple the functionality of the activity from the operational framework. Instead, in many cases, the targeted functionality must be re-implemented if it is required by other systems.

Web services are sometimes presented as the silver bullet of interoperability for distributed computing use cases. PGrade [39] and Taverna [52]

are well known examples of service-based solutions. Yet, using Web services only partially solves the interoperability problem, namely how to in a protocol and programming language independent manner invoke a capability (an operation) offered by a remote entity (a service). Issues related to the coordination of these activities, i.e., to the workflows, including workflow MoC and workflow language are not addressed. Standardization efforts for web service coordination languages, e.g., BPEL [37], have been found unsuitable for scientific workflows [6].

In the Grid, execution level interoperability often means being able to execute activities in resources that use different Grid middlewares. This type of interoperability is a well studied problem with several solutions. For example, in previous work we have show how this can be achieved by decoupling the submission of activities from the control of dependencies in a Grid workflow engine [20]. Then, by using a chain-of-responsibility pattern the correct middleware for executing each activity is selected at run time. A similar solution but at the activity level is presented in [21]. A more specialized solution that also operates with different middlewares and can work with groups of activities while offering fault tolerance is provided by our Grid Job Management Framework (GJMF) [19]. Similar approaches to workflow execution interoperability are proposed by P-Grade [39].

6.4 Granularity concerns

As we have seen from the discussions of conditions and iterations, the granularity of workflow activities affects the performance as well as the complexity of workflows. Too fine granularity can limit the performance due to a higher overhead in Grid interactions. Conversely, having too coarse-grained activities can also reduce workflow performance, in this case due to a reduction in concurrency as the problem can no longer be partitioned into smaller chunks that can operate independently without synchronization. Other issues in which granularity is of concern include the Grid interaction style and whether sub-workflows or individual activities are the basic means of submitting work to the Grid. In general, there is a trade-off between granularity on one hand and complexity and performance on the other. Nevertheless, varying the granularity of activities can be beneficial when striving for interoperability between systems.

6.5 Concluding remarks

In this work we give a comprehensive presentation of the different problems that directly affect interoperability among scientific workflows. Part of our results is the introduction of three dimensions for addressing interoperability issues. The degree of coupling between these dimensions (MoC, language, and execution environment) has interesting consequences. For example, an important lesson learned in our work with a middleware independent Grid workflow engine [20] is that a complete decoupling between execution environment (i.e., Grid middleware and job description language) and workflow language improves portability and interoperability of the engine, but also makes workflow design more tedious and error prone, as the workflow activities, viewed as black boxes by the enactment engine, are completely untyped. There hence exists a trade-off between usability and interoperability.

A similar trade-off also exists between execution environments and MoCs. For example, in essence, the goals of local and Grid workflow MoCs differ significantly. For local workflows, users are better able to express their solutions using MoCs closer to the problem space, as illustrated by the many different dataflow style solutions found for local workflows. However, in these solutions activities are tightly coupled to a particular workflow system and it is not easy to reuse those activities in a different one. On the other hand, in Grid workflows it is simple to provide interoperability at the middleware level. Yet it is harder to specify Grid workflows as, e.g., conditions and iterations are not always available. From our previous discussion we can argue that in local workflows it is better to interoperate at the *workflow level* whereas in Grid workflows is preferred to do so at the *activity level*. Furthermore, from the execution environment dimension, our findings support the use of hierarchical approaches that consider sub-workflows (and all activities) as black boxes.

At the workflow language level a more important trade-off is that between usability and complexity on one hand and potential concurrency (and thus performance) on the other. This trade-off appears in any decision for varying the granularity of activities. The most illustrative case is the different ways in which conditions and iterations are implemented by different systems. While it is possible to translate between languages, differences in implementation details may lead to a tedious processes, performed in ad-hoc ways, and not prone to automation.

Acknowledgments

We are grateful to Frank Drewes and Johanna Högberg for valuable feedback on theoretical aspects of computation. We are also grateful to P-O Östberg for fruitful discussions on general aspects of workflow systems and on workflow language constructs. We thank Ken Klingenstein and Dennis Gannon, organizers of the 2007 NSF/Mellon Workshop on Scientific and Scholarly Workflow, as well as the participants of this important meeting that gave us the opportunity for discussing relevant aspects of interoperability. 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.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, 1986.

[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.136.pdf, February 2009.

[3] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high-performance scientific computing. In *The Eighth International Symposium on High Performance Distributed Computing*, pages 115–124, 1999.

[4] D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)*, 26(4):345–420, 1994.

[5] E.M. Bahsi, E. Ceyhan, and T. Kosar. Conditional workflow management: A survey and analysis. *Scientific Programming*, 15(4):283–297, 2007.

[6] R. Barga and D. Gannon. Scientific versus business workflows. In I. Taylor et al., editors, *Workflows for e-Science*, pages 9–18. Springer-Verlag, 2007.

[7] J. Bentley. Programming pearls: little languages. *Communications of the ACM*, 29(8):711–721, 1986.

[8] A-C Berglund, E. Elmroth, F. Hernández, B. Sandman, and J. Tordsson. Combining local and Grid resources in scientific workflows (for Bioinformatics). In *9th International Workshop, PARA 2008 (accepted)*. Lecture Notes in Computer Science, Springer-Verlag, 2009.

[9] 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 Computat.: Pract. Exper.*, 18(10):1021–1037, 2006.

[10] CoreGRID. Deliverable D.PM.04 basic features of the grid component model. Beta working paper series, wp 47, CoreGRID - Network of Excellence, 2007. Available at: www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf.

[11] P. Couvares, T. Kosar, A. Roy, J. Weber, and K. Wenger. Workflow management in Condor. In I. Taylor et al., editors, *Workflows for e-Science*, pages 357–375. Springer-Verlag, 2007.

[12] W.R. Cowell and C.P. Thompson. Transforming FORTRAN DO loops to improve performance on vector architectures. *ACM Transactions on Mathematical Software (TOMS)*, 12(4):324–353, 1986.

[13] E. Deelman. Looking into the future of workflows: the challenges ahead. In I. Taylor et al., editors, *Workflows for e-Science*, pages 475–481. Springer-Verlag, 2007.

[14] 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*, 25(5):528–540, 2009.

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

[16] T. Delaitre, T. Kiss, A. Goyeneche, G. Terstyanszky, S. Winter, and P. Kacsuk. GEMLCA: Running legacy code applications as grid services. *J. Grid Computing*, 3(1–2):75–90, 2005.

[17] D. Erwin (editor). UNICORE plus final report. www.unicore.eu/documentation/files/erwin-2003-UPF.pdf, visited December 2008.

[18] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity – the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[19] 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.

[20] 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*, pages 259–270. Lecture notes in Computer Science 4967, Springer-Verlag, 2008.

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

[22] W. Emmerich, B. Butchart, and L. Chen. Grid service orchestration using the business process execution language (BPEL). *J. Grid Computing*, 3(3–4):238–304, 2005.

[23] S.D.I. Fernando, D.A. Creager, and A.C. Simpson. Towards build-time interoperability of workflow definition languages. In V. Negru et al., editors, *SYNASC 2007, 9th international symposium on symbolic and numberic algorithms for scientific computing*, pages 525–532, 2007.

[24] International Organization for Standardization. ISO/IEC 2382-1 information technology - vocabulary - part 1: Fundamental terms, 1993.

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

[26] Y. Gil. Workflow composition: semantic representations for flexible automation. In I. Taylor et al., editors, *Workflows for e-Science*, pages 244–257. Springer-Verlag, 2007.

[27] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers. Examining the challenges of scientific workflows. *IEEE Computer*, 40(12):24–31, 2007.

[28] T. Glatard and J. Montagnat. Implementation of Turing machines with the Scufl data-flow language. In *Eighth IEEE International Symposium on Cluster Computing and the Grid*, pages 663–668. IEEE, 2008.

[29] A. Goderis, C. Brooks, I. Altintas, E. Lee, and C. Goble. Heterogeneous composition of models of computation. *Future Generation Computer Systems*, 25(5):552–560, 2009.

[30] 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.

[31] 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.

[32] W.D. Hillis and G.L Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[33] A. Hoheisel. User tools and languages for graph-based Grid workflows. *Concurrency Computat. Pract. Exper.*, 18(10):1001–1013, 2006.

[34] A. Hoheisel and M. Alt. Petri nets. In I. Taylor et al., editors, *Workflows for e-Science*, pages 190–207. Springer-Verlag, 2007.

[35] A. Jantsch and I. Sander. Models of computation and languages for embedded system design. *IEEE Proc.-Comput. Digit. Tech.*, 152(2):114–129, March 2005.

[36] K. Jensen. An introduction to the practical use of coloured Petri nets. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets II: Applications*, pages 237–292. Lecture Notes in Computer Science 1492, Springer-Verlag, 1998.

[37] D. Jordan and J. Evdemon (chairs). Web Services Business Process Execution Language version 2.0. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf, September 2008.

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

[39] P. Kacsuk and G. Sipos. Multi-grid and multi-user workflows in the P-GRADE Grid portal. *J. Grid Computing*, 3(3-4):221–238, 2006.

[40] K.M. Kavi, B.P. Buckles, and U.N. Bhat. Isomorphisms between petri nets and dataflow graphs. *IEEE Trans. on Software Engineering*, 13(10):1127–1134, 1987.

[41] B. Kiepuszewski, A.H.M. ter Hofstede, and C. Bussler. On structured workflow modelling. In B. Wangler and L. Bergman, editors, *Advanced Information Systems Engineering, Proceedings of the 12th International Conference, CAiSE 2000*, pages 431–445. Lecture Notes in Computer Science 1789, Springer-Verlag, 2000.

[42] F. Leymann. Choreography for the grid: towards fitting bpel to the resource framework. *Concurrency Computat.: Pract. Exper.*, 18(10):1201–1217, 2006.

[43] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E.A. Leen, J. Tao, and Y. Zhao. Scientific workflow management and the Kepler system. *Concurrency Computat.: Pract. Exper.*, 18(10):1039–1065, 2006.

[44] Michael Mair, Jun Qin, Marek Wieczorek, and Thomas Fahringer. Workflow conversion and processing in the ASKALON grid environment. In *2nd Austrian Grid Symposium*, pages 67–80. Österreichische Computer Gesellschaft, 2006.

[45] M.W. Margo, K. Yoshimoto, P. Kovatch, and P. Andrews. Impact of reservations on production job scheduling. In E. Frachtenberg and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 116–131. Lecture Notes in Computer Science 4942, Springer-Verlag, 2008.

[46] C. Mateos, A. Zunino, and M. Campo. A survey on approaches to gridification. *Softw. Pract. Exper.*, 38(5):523–556, 2008.

[47] A.S. McGough, W. Lee, J. Cohen, E. Katsiri, and J. Darlington. ICENI. In I. Taylor et al., editors, *Workflows for e-Science*, pages 395–415. Springer-Verlag, 2007.

[48] T. McPhillips, S. Bowers, D. Zinn, and B. Ludäescher. Scientific workflow design for mere mortals. *Future Generation Computer Systems*, 25(5):541–551, 2009.

[49] M. Mosconi and M. Porta. Iteration constucts in data-flow visual programming languages. *Computer languages*, 26:67–104, 2000.

[50] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[51] 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.

[52] T. Oinn, M. Greenwood, M. Addis, M.N. 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 Computat.: Pract. Exper.*, 18(10):1067–1100, 2006.

[53] NSF/Mellon Workshop on Scientific and Scholarly Workflow. Improving interoperability, sustainability and platform convergence in scientific and scholarly workflow. https://spaces.internet2.edu/display/SciSchWorkflow/Home, Visited August 2008.

[54] S.G. Parker, D.M. Weinstein, and C.R. Johnson. The SCIRun computational steering software system. In E. Arge et al., editors, *Modern Software Tools in Scientific Computing*, pages 1–40. Birkhauser press, 1997.

[55] C. Pautasso and G. Alonso. The JOpera visual composition language. *Journal of visual languages and computing*, 16(1–2):119–152, 2005.

[56] C. Pautasso and G. Alonso. Parallel computing patterns for grid workflows. In *Proc. of the HPDC2006 Workshop on Workflows in Support of Large-Scale Science (WORKS06) Paris France*, June 2006.

[57] K. Plankensteiner, R. Prodan, T. Fahringer Attila Kertész, and P. Kacsuk. Fault-tolerant behavior in state-of-the-art Grid workflow management systems. Technical report, CoreGRID, 2007. http://www.coregrid.net/mambo/images/stories/TechnicalReports/tr-0091.pdf, August 2008.

[58] J. Qin and T. Fahringer. Advanced data flow support for scientific grid workflow applications. In *Proceedings of the ACM/IEEE Conference on Supercomputing SC 2007*, pages 1–12. ACM, 2007.

[59] H. Reekie. *Realtime signal processing: dataflow, visual, and functional programming*. PhD thesis, University of Thechnology at Sidney in the School of Electrical Engineering, September 1995.

[60] N. Russell, W.M.P. van der Aalst, and A.H.M. ter Hofstede. newYAWL:designing a workflow system using coloured Petri nets. In N. Sidorova et al., editors, *Proceedings of the International Workshop on Petri Nets and Distributed Systems (PNDS'08)*, pages 67–84, 2008.

[61] M. Siddiqui, A. Villazon, and T. Fahringer. Grid capacity planning with negotiation-based advance reservation for optimized qos. In *Proceedings of the ACM/IEEE Conference on Supercomputing SC 2006*. IEEE, 2006.

[62] W. Smith, I. Foster, and V. Taylor. Scheduling with advance reservations. In *14th International Parallel and Distributed Processing Symposium*, pages 127–132. IEEE, 2000.

[63] Q. Snell, M. Clement, D. Jackson, and C. Gregory. The performance impact of advance reservation meta-scheduling. In D.G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing: IPDPS 2000 Workshop, JSSPP 2000*, pages 137–153. Lecture Notes in Computer Science 1911, Springer-Verlag, 2000.

[64] I. Taylor, M. Shields, I. Wang, and A. Harrison. The Triana workflow environment: architecture and applications. In I. Taylor et al., editors, *Workflows for e-Science*, pages 320–339. Springer-Verlag, 2007.

[65] W.M.P. van der Aalst and A.H.M. Hofstede. YAWL: Yet another workflow language. *Information Systems*, 30(4):245–275, 2005.

[66] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

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

[68] C.Y. Wong, T.S. Dillon, and K.E. Forward. Analysis of dataflow program graphs. In *IEEE International Symposium on Circuits and Systems, ISCAS '98*, volume 2, pages 1045–1048. IEEE, 1988.

[69] Y. Zhao, M. Wilde, and I. Foster. Virtual data language: A typed workflow notation for diversely structured scientific data. In I. Taylor et al., editors, *Workflows for e-Science*, pages 258–275. Springer-Verlag, 2007.

[70] Z. Zhao, S. Booms, A. Belloum, C. de Laat, and B. Hertzberger. VLE-WFBus: a scientific workflow bus for multi e-science domains. In *2nd IEEE international conference on e-Science and Grid computing*, pages 11–19, 2006.

VIII

Paper VIII

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

E. Elmroth, F. Hernández, J. Tordsson, and P-O. 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.

* 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 callout 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.