



GJMF – a composable service-oriented grid job management framework

Per-Olov Östberg*, Erik Elmroth

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

ARTICLE INFO

Article history:

Received 21 April 2010

Received in revised form

20 April 2012

Accepted 29 April 2012

Available online 9 May 2012

Keywords:

Grid computing

Grid job management

Grid ecosystem

ABSTRACT

There exists a number of grid infrastructures in production use for a wide range of scientific applications. However, due to the complexities inherent to construction of distributed computing environments, many grid tools and applications remain tied to specific grids and grid middlewares. In this work we investigate best practices for grid software design and development, and propose a composable, loosely coupled Service-Oriented Architecture for grid job management. The architecture is designed for use in federated grid environments and defines a model for transparent grid access that aims to decouple grid applications from grid middlewares and facilitate concurrent use of multiple grid middlewares. The architecture model is discussed from the point of view of an ecosystem of grid infrastructure components, and is presented along with a proof-of-concept implementation of the architecture.

© 2012 Elsevier B.V. All rights reserved.

1. Introduction

In this work, we extend on prior work documented in [1–3], and investigate best practices for software design in the context of grid job management. As a result, we propose a composable Service-Oriented Architecture (SOA) for grid job management constituted by layers of loosely coupled, composable, and replaceable web services. The proposed architecture is illustrated with a prototype implementation called the *Grid Job Management Framework (GJMF)*.

There are many grid applications and infrastructures in production use today, but as a result of the inherent complexity of grid construction and the interoperability issues of current grid middlewares, many grid applications are tightly coupled to specific grids and grid middlewares. To isolate grid end-users and applications from details of underlying middlewares, and create a more loosely coupled model of grid resource use, job management tools should be designed to operate on top of middlewares, abstract middleware functionality, and offer middleware-agnostic interfaces to grid job management. By decoupling grid applications from grid middlewares, grid job management tools can promote development and migration of computational tools to distributed computing environments, as well as facilitate migration of existing applications to new distributed computing environments.

Grids are highly complex distributed computing environments that exhibit high degrees of heterogeneity in, e.g., resources, software, and administrative practices. In these settings, software

usability becomes a function of factors such as flexibility, scalability, and interoperability, which need to be considered in software design and development. To address these heterogeneity issues, we employ a model of software sustainability based on evolution of software in an ecosystem of grid components (as described in Section 2). Here we identify a set of traits likely to promote survival for job management tools in a grid ecosystem, and explore software design patterns and development methodologies that result in composable software that can thrive (inhabit and interoperate between niches) in such an ecosystem.

Using this ecosystem-based software design model we propose an approach to grid job management based on stratification of job management functionality into a layered service model, and an architecture rendering a set of job management functionality as interoperable web services. The proposed architecture provides a set of middleware-agnostic job management interfaces that aim to allow applications to function seamlessly across grid boundaries, while making concurrent use of multiple grids and grid middlewares.

The architectural model is designed as a framework built on the principle of abstraction; functionality is stratified into layers of services that incrementally provide more advanced functionality through aggregation of underlying service capabilities. Designing software in abstractive layers facilitates component and system simplicity, robustness, and automation. To provide software flexibility the framework defines constructs for dynamic configuration and fault tolerance. To preserve efficiency in service communication and computation, the framework also defines service communication optimization mechanisms that are built into the service model of the framework.

This approach enables developers to build systems where individual components can be deployed stand-alone or as part of other

* Corresponding author. Tel.: +46 907867709; fax: +46 907866126.

E-mail addresses: p-o@cs.umu.se (P.-O. Östberg), elmroth@cs.umu.se (E. Elmroth).

URL: <http://www.cs.umu.se/ds> (P.-O. Östberg).

architectures, while serving as part of the framework. Application developers can select parts of the framework to make use of based on current application needs, and administrators are able to reconfigure framework deployments dynamically. The contributed software prototype provides transparent concurrent access to multiple grid middlewares through a set of job management web services. All services of the prototype provide abstractive views of particular types of grid job management functionality that help to decouple applications from grid middlewares. Throughout the paper, facets of intended system behavior and implications of system design and architecture are discussed.

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

2. Software requirements in the grid ecosystem

An ecosystem can be defined as a system formed by the interaction of a community of organisms with their shared environment. Central to the ecosystem concept is that organisms interact with all elements in their surroundings, and that ecosystem niches are formed from specialization of interactions within the ecosystem. In an ecosystem of grid components [2,4] niches are defined by functionality required and provided by software components, end-users, and other grid actors; and grid applications and infrastructures are constituted by systems composed of components selected from the ecosystem. Here software compete on evolutionary bases for ecosystem niches, where natural selection over time preserves components better at adapting to altered conditions. Adaptability is hence defined in terms of interoperability, efficiency, and flexibility. For software to be successful in the grid ecosystem, individual software components should be composable, replaceable, able to integrate non-intrusively with other components, support established niche actors, e.g., grid middlewares and applications, and promote adaptability through ease of use and administration.

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

Like in any evolution-based system, adaptability and efficiency are key to sustainability in the grid ecosystem. By creating systems composed of small, well-defined, and replaceable components, functionality can be aggregated into flexible applications, resulting in increased survivability for both components and applications [3]. The idea to create composed and loosely coupled applications from aggregation of components is at the core of Service-Oriented Architecture (SOA) [5] methodology.

In the proposed architecture, components are realized as web services that contain customization points where third party plug-ins can be used to alter or augment system and component behavior. To promote deployment flexibility, architecture composition as well as component customization, can be dynamically altered via service configurations as described in [3]. Additionally, individual components of the architecture can be used as stand-alone services, or in other composed architectures, while concurrently serving as part of the architecture in the same deployment environment. For more information regarding grid ecosystem software design and software requirements, see [2,3].

3. Framework service levels and architecture

The practice of developing and deploying infrastructure components as dynamically configured SOAs facilitates development of flexible and robust applications that aggregate component functionality and are capable of dynamic reconfiguration [3]. This approach also provides a model for distributed software reuse, both on component and code level, and facilitates integration software development with a minimum of intrusion into existing systems [6]. Providing small, single-purpose components reduces component complexity and facilitates adaptation to standardization efforts [3].

The architectural model used has previously been briefly introduced in [1], and various aspects of the software development model are discussed in [2,6,3]. The software development model used in this work is a product of work in the Grid Infrastructure Research and Development (GIRD) multiproject [7] and is documented in [2,3]. The models favor architectures built on principles of flexibility, robustness, and adaptability; and aim to produce software well adjusted for interoperability and use in the grid ecosystem [4].

3.1. Architecture layers

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

- **Simplicity.** By hierarchically segmenting the functionality of the framework, components can be kept task-oriented and implemented as small, single-purpose modules that delegate functionality requirements to lower layers. Hierarchical ordering of the architecture also allows simplistic client-server models of communication between components.
- **Robustness.** Simplicity in implementation reduces complexity and increases robustness. Division of functionality into layers and definition of interfaces between components introduce natural failure management points, simplifies failure detection, and allows reactive failure recovery through, e.g., resubmission of tasks.
- **Automation.** Stratification of the architecture into a hierarchy of components allows abstraction of functionality in lower layers and facilitates introduction of automation of framework functionality in higher layers.
- **Flexibility.** Exposing individual components of the framework as services offers clients the choice of what levels of control and automation to make use of. Segmentation of the system into services also increases system deployment flexibility.
- **Efficiency.** Segmentation of system functionality into distributed components introduces communication overhead and synchronization issues. Keeping component segmentation hierarchical minimizes synchronization issues and facilitates parallel (pipeline) processing of framework tasks (masks communication overhead).

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

3.1.1. Grid middleware layer

In the architecture, the grid middleware layer houses all software components concerned with abstraction of native job management capabilities. This includes traditional grid middlewares

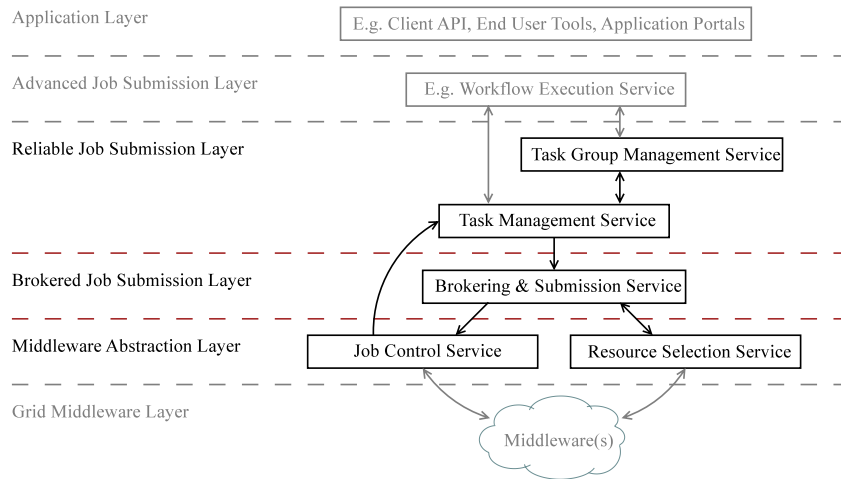


Fig. 1. The proposed framework architecture. Services organized in hierarchical layers of functionality. Within the framework services communicate hierarchically, service clients are not restricted to this invocation pattern.

abstracting batch systems, e.g., the Globus middleware (GT4) [8] abstracting the Portable Batch System (PBS) [9], standardized job dispatchment services, e.g., the OGSA BES [10], and desktop grid approaches, e.g., Condor [11] abstracting use of CPU cycle scavenging and volunteer computing resources. Components in the grid middleware layer are not part of the framework but are essential in providing native job submission, control, and monitoring capabilities to the framework.

3.1.2. Middleware abstraction layer

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

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

3.1.3. Brokered job submission layer

Placed atop of the middleware abstraction layer, the brokered job submission layer provides aggregated functionality for indirect, or brokered, job submission. Services in this layer provides automated matching of jobs to computational resources, and are expected to primarily be used by simple job submission interfaces, e.g., command-line job submission tools.

Job submission performed by services in the brokered job submission layer relies on the targeted job submission capabilities and the information system interfaces of the middleware abstraction layer, and provides a best effort type of failure handling by identifying a set of suitable computational resources for a job and (sequentially) submitting the job to each of these until the job is accepted by a resource. Services in the brokered job submission layer do not provide job monitoring capabilities, as job submission here is expected to result in monitorable jobs in middleware abstraction layer services.

3.1.4. Reliable job submission layer

Intended as the robust job submission abstraction of the architecture, services of the reliable job submission layer provide fault-tolerant and autonomous job submission and management capabilities, and are intended to be utilized (through job management tools) by end-users. The term reliable job submission refers to the ability of these services to handle different types of errors in the job submission and execution processes through resubmission of jobs according to predefined failover policies.

Services in the reliable job submission layer rely on services of the brokered job submission layer for brokering and job submission, and services of the middleware abstraction layer for job monitoring and control. Functionality and policies for failure handling, e.g., for grid congestion and job execution failures, is defined in customizable service plug-ins, and management of sets of independent jobs is provided. Services of the reliable job submission layer also provide monitoring capabilities for jobs and sets of jobs through job management contexts created for all resources submitted here.

3.1.5. Advanced job submission layer

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

3.1.6. Application layer

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

4. The grid job management framework

Implemented as a prototype of the proposed architecture of Section 3, the Grid Job Management Framework (GJMF) is a Java-based toolkit for submission, monitoring, and control of grid jobs. The GJMF is designed as a hierarchical SOA of cooperating web services where framework composition can be altered dynamically and controlled through service configuration and via customization points in services. The grid-enabled web services of the GJMF are implemented and typically deployed using the Globus Toolkit [12], and are compatible with established grid security models such as the Grid Security Infrastructure (GSI) [13]. The GJMF also conforms to the use of a number of web service and grid standards, e.g., the Web Service Description Language (WSDL) [14], the Web Service Resource Framework (WSRF) [15], and the Job Submission Description Language (JSDL) [16]. The service structure of the GJMF is loosely based on the design of the Open Grid Service Architecture (OGSA) [17] specifications for the OGSA Basic Execution Service (OGSA BES) [10], and the OGSA Resource Selection Services (OGSA RSS) [17].

The services in the framework interact by passing messages using either request–response (for, e.g., job submissions) or publish–subscribe (for, e.g., state update notifications) communication patterns. The information routed through the framework travels vertically in Fig. 1, and typically consists of job descriptions passed downwards in task and job submissions, and status update notifications propagated upwards in service state coordination messages. All services maintain state representations as WS-Resources [18], and expose these through service interfaces and WS-ResourceProperties [19], allowing clients to inspect state both explicitly and through subscription to WS-BaseNotification [20] messages.

4.1. Job definitions

To facilitate abstraction of Grid middleware functionality, the GJMF defines three types of job definitions.

- A *job* is a concrete job description, containing all information required to execute a program on a (specified) computational resource. Jobs are in the GJMF processed by the Job Control Service and correspond to unique executions of programs on computational resources. Job descriptions typically consist of a JSDL file specifying an executable program, program parameters, computational resource references, file staging information, and optional JSDL annotations containing job processing hints.
- A *task* is an abstract job description that typically requires additional information, e.g., computational resource references, to be realized as a job. This required information is provided in task to resource matching (brokering). Note that by this definition, a job is a task subtype. This allows jobs to be submitted as tasks in the GJMF, in which case any additional brokering information is utilized in the brokering and job submission process. Tasks are in the GJMF processed by the Task Management Service.
- A *task group* is a set of independent tasks (and jobs) that can be executed in any order. Task groups are distinguished from jobs and tasks by having a shared execution context for all tasks in a task group. The processing result of a task group is determined by the combined processing results of the task group's tasks. Task groups are in the GJMF processed by the Task Group Management Service.

Note that the GJMF does not specify a task type that models dependencies between tasks. While there exists a number of

computational applications that define such dependencies (e.g., sequence of concurrency requirements), this is not covered by the GJMF task processing model. The GJMF aims to provide a foundational task execution platform that can be built upon by other more specialized tools that better capture inter-task relationships and dependencies (as illustrated in Fig. 1).

4.2. Components

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

4.2.1. Log Accessor Service (LAS)

In a distributed architecture managing multiple synchronized states, ability to track state development is highly desirable. The Log Accessor Service (LAS) is a service that provides database-like interfaces to job, task, and task group logs generated by the GJMF. Within the GJMF, the LAS is used to record state transitions and job submission and processing information. The LAS utilizes unidirectional data transfers, e.g., the GJMF services use the LAS to store data, and service clients use it to inspect details of task processing.

The LAS maintains internal storage queues and resource serialization mechanisms to minimize overhead for use of the service and provide an asynchronous log storage model. Customizable database support is provided through use of database accessor plug-in modules. Database accessors can be provided by third parties and boiler-plate solutions for accessor plug-ins supporting SQL and JDBC are provided. Currently, the LAS provides accessors for MySQL, PostgreSQL, and Apache Derby. Unlike other services of the GJMF, use of the LAS is optional and not required for any part of the GJMF to function. Both the LAS and LAS database accessors can be configured through the LAS configuration.

4.2.2. JSDL Translation Service (JTS)

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

The JTS employs a modularized architecture where translation semantics are provided by plug-ins. Support for new languages can be added by third parties without modification of the framework. Currently, the JTS supports translation between JSDL [16] and Globus Toolkit 4 Resource Specification Language (GT4 RSL) [12], NorduGrid Extended Resource Specification Language (XRSL) [21],

and a custom dialect of XRSL presented in [6]. Translations of job descriptions are made based on the context of the job description representation created. Typically this means that existing job descriptions are queried for information required to create new representations of corresponding semantics. Type-specific data representations are translated based on the semantics of the enacting middleware, e.g., Uniform Resource Locators (URLs) are reformatted and supplied suitable protocol tags to match middleware transfer mechanism preferences. The JTS can be configured to use a specific set of translation modules, which can be configured through the JTS configuration.

4.2.3. Job Control Service (JCS)

The purpose of the Job Control Service (JCS) is to provide a uniform and middleware-transparent job submission and control interface. The JCS defines a set of generic job control functionality, as well as a job state model (illustrated in Fig. 4(c)), that provide a fundamental view of job management that other services in the GJMF build upon. Within the GJMF, the JCS is used by the Brokering and Submission Service for job submission, and by the Task Management Service for job monitoring and control. Service clients can use the JCS directly as a targeted grid job submission and control tool.

Internally, the JCS maintains a job control component that coordinates execution and monitoring of native grid jobs. Job resources are used to maintain job state and are exposed as inspectable *WS-ResourceProperties* to service clients. The job controller abstracts the use of middleware-specific job dispatcher and dispatcher prioritizer plug-ins, and both the job controller and the middleware dispatchers utilize LASs for log storage. Middleware support in the JCS is provided through customizable and configurable plug-in modules that allow third parties to develop and deploy support for proprietary job management solutions. Middleware dispatchers abstract use of grid middlewares and employ the JTS and the LAS for job description translation and log storage respectively. The JCS currently provides middleware support for the NorduGrid ARC [21], GT4 [8] middlewares. For test and service client development purposes, the JCS also provides a simulation environment where jobs are simulated rather than submitted and executed. This utility allows JCS clients to encounter exotic job behaviors on demand via discrete-event simulation of job state transitions.

The JCS can be configured to use a specific set of middleware dispatchers, a middleware dispatcher prioritizer, a state monitor, a set of JTSs, and an optional set of LASs. The functionality of the JCS can also be altered by providing processing hints through annotations in the JSDL job description. These annotations can affect, e.g., middleware dispatcher prioritization, or provide job submission parameters such as queue system information for ARC submissions (an example from [6]) or GT4 Globus Resource Allocation Manager (WS-GRAM) parameters for Condor-G [22] submissions. As these types of processing hints are completely orthogonal to standard service behavior, i.e. does not affect processing of other jobs or service functionality, they can be used to temporarily alter service behavior for a specific job without alteration of framework composition or configuration.

4.2.4. Resource Selection Service (RSS)

Built on the OGSA RSS [17] model, the GJMF Resource Selection Service (RSS) provides a service interface for resource brokering in grid environments. Within the GJMF, the RSS is used by the Brokering and Submission Service as an execution planning and brokering tool. Service clients can use the RSS directly for job to resource matching or to inspect resource availability.

Internally, the RSS maintains a resource selector component that coordinates brokering of tasks to computational resources.

Middleware-specific information system accessors are used to abstract middleware information systems and provide translations of middleware-specific record formats to an internal RSS format. The RSS also maintains mechanisms for resource information retrieval and caching, information system monitoring, and customization mechanisms that allow third parties to develop plug-ins to support new information sources. In essence, the RSS maintains a cached view of the resource state of known resources, and on demand ranks resources that meet job requirements based on prior performance. As the RSS information state is maintained in background worker threads, the RSS is able to respond to resource selection requests efficiently (as demonstrated in [23]).

The RSS can be configured to retrieve information from a range of information systems, currently including the ARC and GT4 grid middleware information systems, as well as a simulated information system configurable through the RSS configuration intended for service development purposes. The RSS also provides boiler-plate solutions for data access and type conversion to facilitate implementation of custom information accessors.

4.2.5. Brokering and Submission Service (BSS)

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

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

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

4.2.6. Task Management Service (TMS)

The Task Management Service (TMS) provides an interface for automated and fault-tolerant task management, and defines a task state model (illustrated in Fig. 4(b)). The TMS maintains inspectable state contexts for tasks and employs a model of event-driven state management powered by the JCS state mechanisms. To provide failover capabilities, tasks submitted through the TMS are repeatedly submitted and monitored by the TMS until resulting in a successful job execution, or a configurable amount of attempts are made. Within the GJMF, the TMS is used by the Task Group Management Service for management of individual tasks.

Internally, the TMS maintains components for task management, task submission, and job monitoring, as illustrated

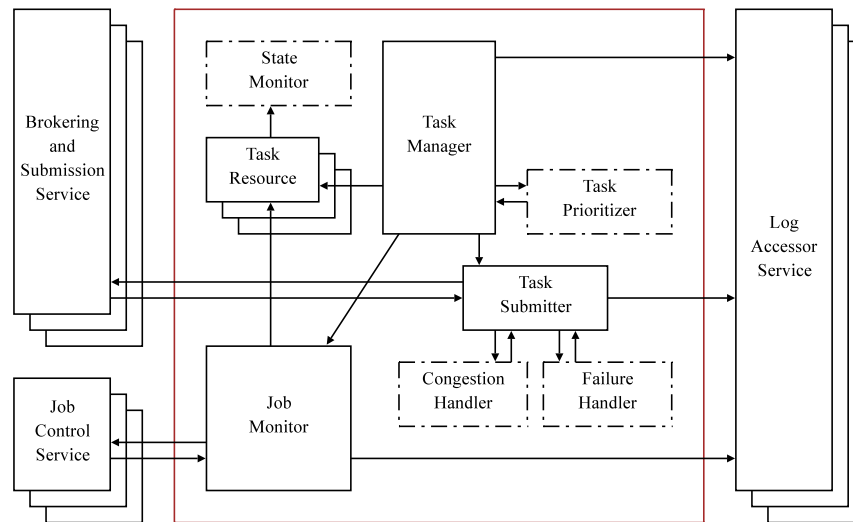


Fig. 2. TMS architecture. The TMS architecture is typical for all GJMF services; interaction with other services are wrapped in access modules, and customization points are exposed through configurable plug-in structures. Customization points are illustrated using dotted lines.

in Fig. 2. Task state is maintained and exposed through WS-ResourceProperties by task resources. The internal mechanisms of the TMS can be customized via configuration and a set of plug-in modules that control task prioritization, congestion handling, failure handling, and state monitoring. To enforce user-level isolation and fair competition in multi-user scenarios, the TMS maintains separate task queues for each user. The TMS relies on the BSS for submission of tasks to grid resources, and can be configured to use customized congestion and failure handlers to control task resubmission behaviors, and a customized task prioritizer to influence task processing order. The TMS can also be configured to use a state transition monitor for event-driven state monitoring, a set of BSSs, and an optional set of LASs.

4.2.7. Task Group Management Service (TGMS)

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

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

The TGMS also provides a mechanism for suspension of (processing of) task groups, a mechanism designed to adapt to scenarios where user credentials expire or large task groups need to be paused. Once suspended, task groups are not further processed until explicitly resumed. Tasks in a suspended task group that are submitted to a TMS will be processed if possible, but no new task submissions are made until the task group is resumed. The TGMS can be configured to use a congestion handler to customize back-off behaviors in grid congestion scenarios; task

group and task prioritizers to customize processing order of task groups, tasks, and jobs; a state transition monitor for event-driven state monitoring; a set of TMSs; and an optional set of LASs.

4.2.8. Common library

The GJMF common library is a service development utility library that provides a common type set and boiler-plate solutions for, e.g., local call optimizations, service stubs, notification subscriptions, credentials delegation, security contexts, worker threads, state management, service client APIs, dynamic configuration, and resource serialization.

The GJMF common library provides a simple framework for service development that defines a service structure used by all services in the GJMF. The service structure is illustrated in Fig. 3, and details separation of service interface implementation from service back-end implementations, and service clients from service client factories. Service client factories are exposed to applications and dynamically instantiate service client implementations based on type of service invocation to be used. Service clients marshal data and perform service invocations, in the case of regular service clients through web service SOAP messages and through direct service back-end invocations using immutable wrapper types for local call optimization clients. Service interface implementations marshal SOAP data through stubs into immutable wrapper types and invoke corresponding methods in service back-ends. Service back-end implementations are responsible for maintaining state in service resources, which are accessed through service resource homes. The service structure of the GJMF common library has previously been discussed in [3].

The service development framework, in concert with the GJMF client API, handles all service invocation mechanics, including data type marshaling, service instantiation, and notification management. The framework encapsulates a local call optimization mechanism that allows service components to be exposed as local objects to other services co-deployed in the same service container, i.e. allowing co-hosted services to make marshaled in-process Java calls directly between service clients and service back-ends. This optimization mechanism, which is discussed in Section 5.1, and also addressed in [3], is made fully transparent to service clients by the service structure of the common library and the client API. As also described in [3], the common library provides a set of basic and immutable types for use in the GJMF client API as well as a type marshaling mechanism that abstracts the use of stub types in the GJMF.

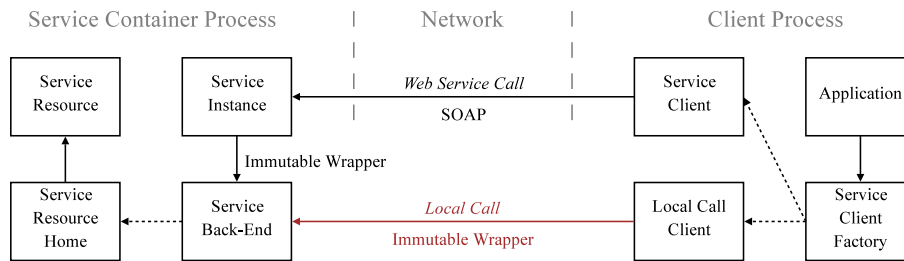


Fig. 3. The GJMF service structure. The GJMF common library provides boiler-plate solutions for service instantiation, service back-end implementation, resource management, and client APIs. The GJMF client API abstracts use of the service invocation optimizations through use of service client factories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

4.2.9. Client application programming interface

The GJMF client Application Programming Interface (API) is a set of Java classes abstracting the use of the GJMF web services for Java programmers. Mimicking the interface of the GJMF services, the client API is designed to provide intuitive use of the framework to developers with limited experience of web service development. All GJMF functionality is accessible through both the GJMF services and the GJMF client API.

5. Architecture discussion

The hierarchical architecture of the GJMF is intended to provide clients a versatile and flexible set of job management interfaces that offer an increasing range of automation of the job management process without sacrificing user control. Services in lower layers offer fine-grained job management interfaces with explicit control, while services in higher layers attempt to automate the job management process and offer control through configuration of behavior and optional customization point modules.

To meet the flexibility and adaptability requirements discussed in Section 2, we extend the software development model previously presented in [3]. Key approaches in this model include use of Service-Oriented Architectures (SOAs) [5], design patterns, refactorization methods, and techniques to improve software adaptability such as dynamic configuration and customization points. Software is developed in Java and the Globus Toolkit [12] is employed to produce grid-enabled web services compatible with established grid security models.

5.1. Invocation patterns

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

When service clients are co-deployed with GJMF services, i.e. deployed in the same service container as the GJMF, service invocations are by default routed through the GJMF local call optimizations. These mechanisms exploit that services hosted in the same container share the same process space, i.e. operate in the same Java Virtual Machine (JVM), and allow service clients to directly invoke methods in service implementation back-ends. By bypassing message serializations, this greatly reduces service

invocation makespans and memory footprints, allow more fine-grained service communication models, and promotes a model of service aggregation where constituent services can function as local objects in aggregated services [3]. In the GJMF this results, e.g., in a reduced need for polling to maintain distributed state coordination as state update notifications are less likely to be lost. All services of the GJMF are designed to be distributed in separate service deployments, but are for performance reasons recommended to be co-deployed.

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

5.2. Deployment scenarios and use cases

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

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

As natural overlaps between these usage scenarios exist, each of these are expected to be seen in hierarchical or other types of federated grid environments, as well as in federated cloud computing systems. Typical usage scenarios for the GJMF are expected to include combinations of multiple deployments of the framework, on top of multiple grid middlewares and resource managers. To meet advanced application requirements, e.g., workflow enactment or parameter sweeps, the GJMF is expected to be utilized in combination with high-level tools such as the Grid Workflow Execution Engine (GWEE) [24] or the StratUm toolkit [25].

An example intended use case for the GJMF is illustrated in [25] where StratUm, an advanced task manager that models dependencies between tasks as well as manages user data for experiments, is placed on top of the GJMF as an integrated framework client. In this scenario, the GJMF acts as a job execution engine that brokers, submits, monitors, and controls jobs in a production grid environment. Use of the GJMF here allows the StratUm toolkit to abstract job management complexity and focus on providing additional functionality on top of the framework.

Table 1
GJMF state interpretations.

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

Another, more infrastructure-oriented use case of the GJMF is demonstrated in [6]. Here the GJMF is deployed as a resource-abstracting back-end to the LUNARC application portal, and customized extensions to the framework is deployed to facilitate portal integration of middleware-agnostic job control and monitoring.

The GJMF deployment flexibility allows the framework to be employed in a number of computational settings, including high-performance computing (HPC) (requiring support from underlying middlewares for some functionality, e.g., MPI job execution), high-throughput computing (HTC), as well as the more recently defined many-task computing (MTC) [26] paradigm. In MTC, focus is placed on enactment of loosely coupled applications constituted by large numbers of short-lived, data intensive, heterogeneous tasks with high (non-message passing) communication requirements, a setting envisioned in the design of the GJMF.

5.3. State models

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

The GJMF state update mechanisms are built on WS-Base-Notifications messages. To compensate for state notifications being dropped due to network failures or service container loads, all services implement a state monitoring mechanism that regularly polls for missing notifications. As both state coordination and monitoring mechanisms are encapsulated in the framework service structures and client APIs, service implementations consider state delivery transparent and reliable.

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

5.4. Data management

To maintain middleware transparency, the GJMF does not actively participate in data transfers. The GJMF assumes that data files are available and can be transferred to and from computational resources by enacting grid middlewares via file transfer mechanisms chosen by the middlewares. File staging

information is conveyed as part of job descriptions, typically in the form of GridFTP [27] URLs, and JSDL annotations can be used to provide job brokering hints related to storage requirements for computational elements. Data files are expected to be available prior to job submission (i.e. the GJMF does not verify the existence of data files during brokering), and computational resources and clients are responsible for maintaining file system allocations capable of accommodating incoming and outgoing data files respectively.

Data transfer URLs are translated by the JTS to formats recognized by the underlying middleware as part of the job description translation process. If the underlying middleware does not support file staging, the JCS customization points can be used to provide data transfer capabilities as part of the middleware job submission process without coupling GJMF clients or services to underlying middlewares. Plans to extend the GJMF with utility mechanisms and services for data management are under investigation, see Section 7.

5.5. Resource brokering

To decouple the GJMF services from grid middlewares and each other, all job to computational resource brokering activities are in the GJMF abstracted by the RSS, which in turn relies on grid middleware information systems for monitoring of computational resource availability, characteristics, and load. As middleware information systems typically contain large volumes of cached information, and grid environments are likely to contain multiple concurrent job submission and management systems, such brokering components will always operate on information that is to some extent deprecated [28].

The RSS is limited to provide computational resource recommendations (execution plans) without feedback from service clients. This abstraction implies that the RSS is agnostic of whether a particular execution plan is enacted or not. To compensate for middleware information system update latencies, it would be possible to maintain an internal cache of prior execution plans and update resource load weights through speculation. As the RSS enforces user-level isolation of service capabilities, a unique cache would be created for each user and restricted to contain recommendations for that user.

To improve quality of resource brokering, it would be possible to interface the RSS with grid accounting and load balancing systems, e.g., the SweGrid Accounting System (SGAS) [29], as well as provide the RSS with feedback from the JCS or job submission systems such as the Job Submission Service (JSS) [28]. To reduce system complexity and maintain a clean separation of concerns, the RSS does not implement speculative resource load prediction, but it does offer customization points for third party implementation of advanced brokering algorithms where such feedback loops can be implemented without affecting the design of the framework.

The current implementation of the RSS is to be regarded a prototype, we foresee development of additional RSS versions with resource selection capabilities of particular interest for certain users [30]. Evaluation of RSS brokering performance and quality of execution plans is considered out of scope for this work.

5.6. Security

The GJMF employs the Grid Security Infrastructure (GSI) [13] security model provided by the Globus Toolkit [12], and can be configured to use Secure Message, Secure Conversation, or Credentials Delegation (i.e. use of the Globus Delegation Service) communication mechanisms. Client and service security modes are individually configured using security descriptors, and service

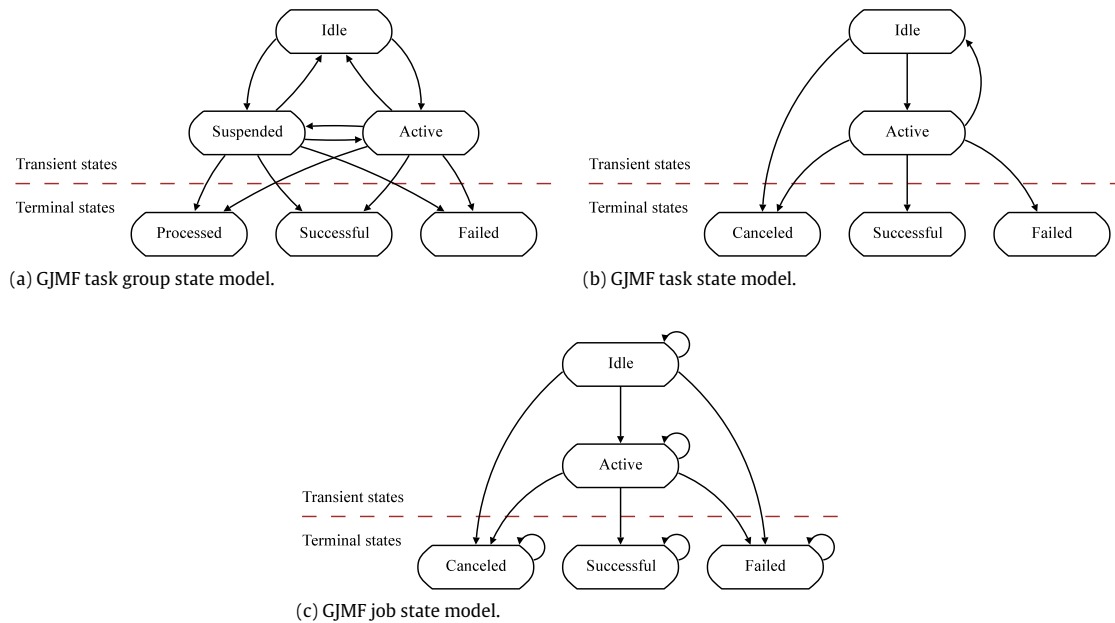


Fig. 4. GJMF state models. Task group states are used in the TGMS, task states in the TGMS and the TMS, job states in the JCS. The JCS job state model is based on the state model of the OGSA BES [10]. Recurring states in the GJMF job state model abstract state information from more fine-grained grid middleware job state models.

client identities are verified for all service invocations from external clients. For service invocations using GJMF local call optimizations, i.e. from co-deployed service clients, credential proxies are accepted without verification of caller identity. This relaxation of authentication is done for performance reasons and is deemed acceptable for situations where mutual trust is established between services and deployment environments. GJMF local call optimizations can be disabled or replaced with Apache Axis local call optimizations for situations where verification of caller identity is required to be enforced.

All types of job definitions, including task groups, are upon submission associated with a set of user credentials used for, e.g., user authentication, resource ownership, and job execution privileges. User credentials are inherited in subsequent submissions within the GJMF, i.e. task group credentials are assigned to tasks upon submission to a TMS, and jobs are assigned task credentials when submitted to a JCS. Task groups distinguish themselves from jobs and tasks by the ability to be suspended in execution, e.g., upon expiration of task group credentials.

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

To facilitate construction of job submission proxies, a requirement in, e.g., grid portal construction [6], clients may specify separate execution credentials for tasks. GJMF resources (including LAS logs) resulting from such submissions are owned by the execution credentials identity. Authentication of caller credentials is performed in GJMF service invocations, but authentication of execution credentials may be deferred (by use of local call optimizations) until grid middleware job submissions.

5.7. Performance characteristics

While a detailed performance analysis is out of scope for this contribution, brief remarks regarding performance characteristics of the framework from [23,25] are included to motivate use of the framework. Implementation of the framework as a network of web services introduces overhead for service invocation, which can be

mediated using invocation optimization mechanisms and masked by parallel processing of framework tasks.

Web service invocation throughput is typically limited by two factors; service computational complexity and service invocation overhead. In the GJMF, services are designed to have low computational complexity, and framework invocation overhead is addressed by use of service invocation optimization mechanisms. As illustrated in Fig. 5(a), the BSS and JCS are designed to make synchronous invocations to underlying systems, and are limited by the job submission performance of grid middlewares. Other services of the GJMF employ asynchronous invocation processing models and do not suffer this limitation. To address invocation overhead issues, the GJMF employs two invocation optimization mechanisms; local calls and batch invocations.

To analyze performance of the framework prototype and evaluate impact of architecture design on system overhead, extensive job submission and processing performance tests have been undertaken [23]. In tests, framework performance regarding service invocation capability (with and without invocation optimizations), job submission throughput, and job processing throughput (under ideal and realistic settings) have been quantified against baseline performance measurements of underlying grid middlewares. All tests have been performed under realistic operational settings, using WS-Security SecureConversation and production use deployments of GT4, Torque, and Maui.

Results indicate that total system overhead can be subdivided into three components: submission overhead, processing overhead, and job execution overhead. Submission overhead is incurred during submission of tasks to the framework, and typically ranges from 1 to 30 s per invocation depending on factors such as class loading overhead (for Java-based service clients), encryption overhead (varies with security model), and transmission overhead (mainly consisting of network latency and XML message serialization overhead). Impact of submission overhead can be mediated by use of batch invocation modes (that reduce the number of invocations) and use of asynchronous processing mechanisms (that reduce service response time). In tests (using large task groups and full WS-Security), effective submission overhead contributions are reduced to fractions of seconds per job [23].

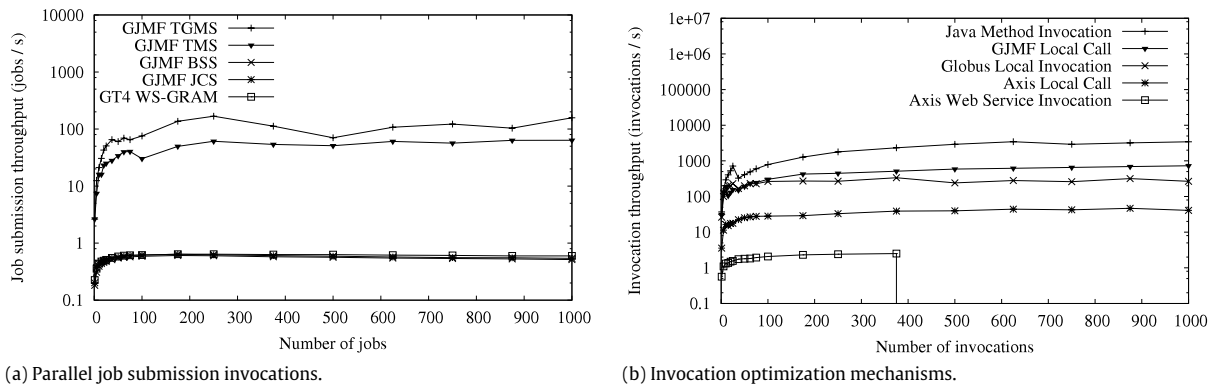


Fig. 5. Service invocation throughput as a function of number of invocations. Vertical axes logarithmic. Illustrations from [23].

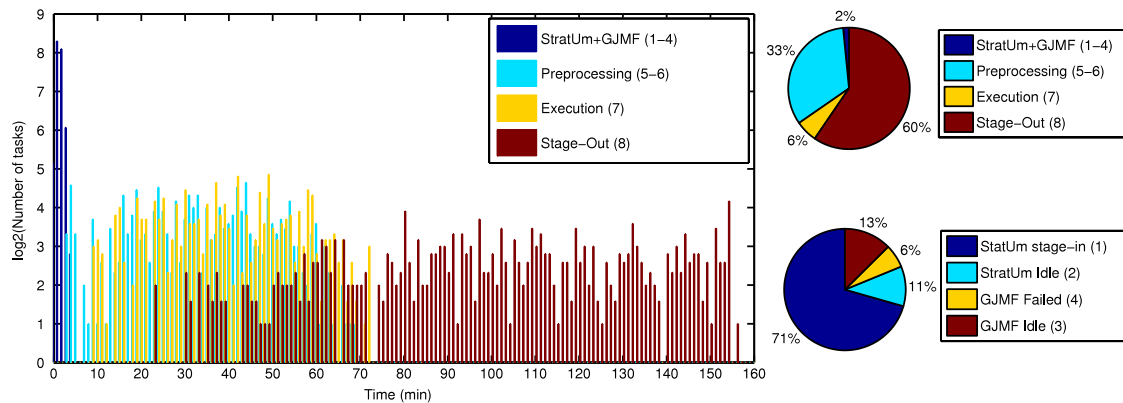


Fig. 6. Relative time distributions for job execution phases in a production environment experiment using the GJMF. Illustration from [25].

Processing overhead consists of service computation and invocation overhead. When framework services are co-deployed, invocation overhead can be reduced to the order of milliseconds by use of local call optimizations. Services using synchronous processing models (BSS and JCS) are limited by performance of underlying systems and produce the largest contributions to total system overhead. Concurrent use of local call optimizations and asynchronous invocation processing models reduce total service invocation overhead contributions to the order of milliseconds. As illustrated in Fig. 5(b), the performance of GJMF local call optimizations are found competitive when compared to similar mechanisms in GT4 and Apache Axis. As the invocation performance of the local call optimizations is close to that of Java method invocations, the internal communication performance of the framework approaches the efficiency of monolithic architectures [23].

Job execution overhead is here defined to include factors such as middleware submission overhead, scheduling overhead, staging and execution makespan, i.e. external factors outside of framework control. As illustrated in Fig. 6, job execution overhead is typically several orders of magnitude larger than GJMF contributions to system overhead in realistic scenarios. In this illustration, which in the histogram details the time spent in different execution phases of jobs, the majority of time is spent performing file staging (preprocessing and stage-out in the illustration) and computations. The overhead imposed by the GJMF (and in this case the StratUm framework presented in [25]) is imposed early and is of very limited scope even when including time spent on failed job executions as GJMF overhead. In the illustrated experiments, which utilize the GJMF as a job management tool in a production environment and run thousands of jobs with relatively short execution time (between 15 min and 4 h) and a low job failure rate (ca 0.5%), the GJMF incurred an average of 12 s of overhead per job when including the cost of failed jobs in GJMF overhead estimates [25].

In summary, total overhead introduced by use of the GJMF for, e.g., job submission, brokering, and monitoring, can be limited to less than one second per job for realistic usage scenarios. When including costs of resubmission and processing of failed jobs in overhead estimations, overhead incurred will scale with the probability of job failure. For stable production environments, such overhead will on average remain in the order of seconds per job. Impact of framework overhead on system performance is mitigated by mechanisms such as asynchronous and parallel processing of tasks, batch invocation modes, and local call optimizations. For further and detailed performance evaluation information, see [23,25].

6. Related work

A number of contributions that in various ways relate to the job management architecture proposed in this work have been identified. Standardization efforts such as JSDL [16], GLUE [31], OGSA BES [10], and OGSA RSS [17] have helped shape boundaries between niches in the grid infrastructure component ecosystem, and directly impacted the design of the proposed architecture. Standardized web service and security technologies such as WSRF [15], WSDL [14], SOAP [32], and GSI [33] have outlined the architecture communication models, and grid middleware and resource manager systems such as the Globus middleware [8], NorduGrid ARC [21], Condor [11], and the Berkeley Open Infrastructure for Network Computing (BOINC) [34] have all contributed to the design of the architecture's middleware abstraction layer. Standardization and interoperability efforts such as The Open Grid Services Architecture (OGSA) [17], the Open Middleware Infrastructure Institute (OMII Europe) [35], and Grid Interoperation/Interoperability Now (GIN) [36], as well as

contributions such as [37–40] have provided perspective, insight, and inspiration to architecture interoperability design.

The grid resource management system survey presented in [41] provides a taxonomy of grid job management systems. In this model, the GJMF is classified as a job management system providing soft quality of service for computational grids. Resource organization, namespace, information system, discovery, and dissemination as defined in this model are all determined by the underlying middleware. Type of scheduler organization is determined by how the framework is employed, but is typically expected to be decentralized for multi-user use of the framework. Non-predictive state estimation models are currently provided by the RSS, along with event-driven and extensible (re)scheduling policies.

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

The GridWay Metascheduler [42], a framework for adaptive scheduling and execution of grid jobs. Like the GJMF, GridWay builds on the Globus Toolkit and offers an abstractive type of grid job submission focused on reliable and autonomous execution of jobs. Both systems provide failover capabilities through resubmission of jobs, where GridWay offers job migration capabilities through checkpointing and migration interfaces, whereas the GJMF focuses on abstraction of grid middleware capabilities and system composability, and offers coarse-grained resubmission policies in higher services. GridWay also offers a performance degradation mechanism which may be used to detect and trigger job migration mechanisms. The GJMF assumes computational hosts maintain consistent performance levels and relies on grid applications and middlewares to handle checkpointing and application preemption issues.

The Falcon [43] framework provides a fast and lightweight task execution framework focused on task throughput and efficiency. Falcon is by design not a fully featured local resource manager, and achieves high job submission throughput rates through, e.g., elimination of features such as multiple submission queues and accounting, and the use of custom protocols for state updates. Both Falcon and the GJMF are service-based frameworks and make use of notifications for distributed state notifications, but are in essence designed for different use cases. Falcon is, e.g., designed for efficient job submissions and achieve much higher submission throughput than the GJMF, whereas the GJMF, e.g., provides middleware-transparency to service clients.

The Minimum intrusion Grid (MiG) [44] is a framework aimed at providing grid middleware functionality while placing as little requirements as possible on grid users and resources. Building on existing operating system and grid tools such as SSH and X.509 certificates, the MiG provides a non-intrusive integration model and abstracts the use of grid resources through service-based interfaces. The approaches differ on a number of points, e.g., where the MiG uses a centralized and monolithic job scheduler the GJMF provides a framework of composable services and abstracts use of grid middlewares.

The Imperial College e-Science Networked Infrastructure (ICENI) [45] is a composable OGSA grid middleware implementation based on Jini. ICENI provides a semantic approach to build autonomously composable grid infrastructure components where services are annotated with capability information and new services are instantiated through SLA negotiations with existing services. The ICENI composability approach differs from the GJMF one, whereas the GJMF only provides mechanisms for framework (re)composition and service customization. ICENI also exposes service implementations locally through the Jini registry, a mechanism similar to the GJMF local call optimizations, and provisions for plug-in implementations of schedulers and launchers [46] in a way similar to the GJMF RSS customization points. Compared to ICENI,

the GJMF provides additional functionality in terms of higher-level abstractions of job management, client APIs, more flexible deployment options, and greater standardization support.

The Job Submission Service (JSS) [28] is a resource brokering and job submission service developed in the GIRD [7] project. The JSS supports advanced brokering capabilities, e.g., advance reservation of resources and co-allocation of jobs, customization of algorithms through plug-ins, and standards-based middleware-agnostic job submission. Compared to the JSS, the GJMF provides additional functionality in, e.g., management and monitoring of jobs and groups of jobs, client APIs, logging capabilities, translation of job descriptions, and incorporation of more recent standardization efforts. Work on the GJMF builds on experiences from the JSS project.

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

eNANOS [47] is a resource broker that abstracts grid resource use and provides an API-based model of grid access. Internally, uniform resource and job descriptions combined with XML-based user multi-criteria descriptions provide dynamic policy management mechanisms facilitating use of advanced brokering mechanisms. Job and resource monitoring mechanisms are provided, and failure handling through resubmission of jobs is supported. The primary differences between eNANOS and the GJMF lie in the flexibility of the GJMF architecture, which provides dynamic composition of the framework and additional levels of abstraction of job management functionality. The GJMF also builds on standardization efforts such as JSDL, WSRF, and the OGSA BES.

The Community Scheduler Framework (CSF4) [48] is an OGSA-based open source grid meta-scheduler. Like the GJMF, CSF4 is constructed as a framework of web services, builds on GT4, provides WSRF compliance, and exposes abstractions for job submission and control. In addition, CSF4 also provides user-selectable job submission queues and a mechanism for advance reservation of resources (via local resource managers). Compared to the CSF4, the GJMF provides support for concurrent use of multiple middlewares, framework composability, standards compliance, and a Java-based client API.

The GridLab Grid Application Toolkit (GAT) [49] is a high-level toolkit for grid application development. The fundamental ideas behind the GAT and the GJMF are similar, both aim to decouple grid applications from middlewares by providing middleware-agnostic grid access through client APIs aimed at simplifying grid application development. The GAT builds on the GridLab [50] architecture which aims to be a complete grid utilization platform, providing, e.g., data management services (including data transfer and replica management capabilities), monitoring services, and services for visualization of data, while the GJMF provides a composable and lean architecture for grid utilization focusing on job management functionality, and relying on underlying middlewares for job control and file staging capabilities.

GridSAM [51] is a standards-based job submission system that builds on standardization efforts such as JSDL, and aims to provide transparent job submission capabilities independent of underlying resource managers through a web service interface. Similar to the asynchronous job processing of the GJMF, GridSAM employs a job submission pipeline inspired by the staged event-driven architecture (SEDA) [52] that allows for short response times in job submission. Fault recovery capabilities are in GridSAM built by persisting event queues and job instance information, similar to the failure handling mechanisms of the GJMF that provide redundancy and resubmission capabilities. Compared to GridSAM, the GJMF provides additional functionality for composition of the framework, job description translation functionality, job monitoring capabilities, and multiple job submission and control mechanisms.

Nimrod-G [53] provides a layered architecture for resource management and scheduling for computational grids. Nimrod-G provides an economy-driven broker that supports user-defined deadline and budget constraints for schedule optimizations [54], and manages supply and demand of resources through the Grid Architecture for Computational Economy (GRACE) [55]. Like the GJMF architecture, the Nimrod-G provides layered abstractions of middleware access components and facilitates use of parameter-sweep style applications. While the GJMF lacks capabilities for economy-based scheduling decisions, it offers customization points for these types of mechanisms in the RSS, and provides a flexible architecture that can incorporate usage pattern-specific adaptations with only local modifications.

The Gridbus [56] broker is a grid broker that mediates access to distributed data and computational resources, and brokers jobs to resources based on data transfer optimality criteria. Gridbus extends the resource broker model of Nimrod-G, defining a hierarchical model for job brokering containing separate resource discovery, grid scheduling, and monitoring components. Like in the GJMF, tasks are defined as sequences of commands that describe user requirements, including, e.g., file staging and job execution information, located within the task description itself. Task requirements drive resource discovery and tasks are resolved into jobs, here defined as units of work sent to grid nodes, i.e. instantiations of tasks with unique combinations of parameter values. The Gridbus broker also abstracts use of multiple middlewares through a service-based interface. Differences between the two platforms include, e.g., Gridbus heuristics-based scheduling strategies, and the GJMF's ability to reconfigure framework deployments.

GMarte [57] is a grid metascheduler framework exposing a high-level Java API for grid application development. Like the GJMF, the GMarte architecture is built in layers and employs a middleware abstraction layer that abstracts use of multiple middlewares. GMarte also provides failure handling through resubmission of jobs, and extends upon this through provisioning for application-level checkpointing of job executions. GMarte exposes a Java client API, plug-in points for information system access, and a service-based interface through GMarteGS [58], which supports WS-BaseNotification based state updates. The GJMF differs from the GMarte on a number of points, e.g., through the use of standardization efforts like JSDL and the OGSA BES, and by providing a dynamically composable architecture.

The Grid Meta-Broker Service (GMBS) [59] addresses grid job management in a way similar to the GJMF. The goals of both projects include to provide interoperability between grids through automation and virtualization of job management without modification of grid middleware deployments. The GMBS defines an architecture for interoperability on a meta-broker level, defines languages for meta-broker scheduling and broker description, and performs brokering of jobs to resource brokers in a way similar to how the GJMF performs brokering of jobs to resources. Both architectures expose functionality through WSDL-based web service interfaces, build on standardization efforts, define translation components for JSDL documents, and provide broker-/resource-specific invocation components. While the GJMF and GMBS are similar in design, concept, and goals, they operate on different levels in the grid job management stack. The GMBS provides grid interoperability through high-level meta-brokering, whereas the GJMF defines a flexible architecture that provides interfaces to multiple types of job management functionality. With modification, the GMBS could make use of the lower layers of the GJMF for resource brokering, and the higher layers of the GJMF could be adapted to make use of the GMBS for job management.

All of these contributions are considered to operate on a layer higher than the grid middleware layer in the GJMF

architecture, and are as job management solutions considered alternative approaches to the GJMF. Each system could naturally be incorporated with the GJMF as grid middleware accessors, or could with modifications utilize the GJMF in a similar manner. Furthermore, there exists a number of workflow-based approaches to grid job management, e.g., ASKALON [60], Pegasus [61], and GWEE [24]. These have been omitted here as the scope of this work is restricted to generic job management architectures. Naturally, with modifications, most of these could make use of the GJMF for middleware-transparent grid access.

Finally, a few slightly different approaches have been identified, e.g., P-GRADE [62], which is a high-level environment for transparent enactment of parallel and grid execution of applications. P-GRADE abstracts use of grid resources through Condor and Globus interfaces, and provides enactment of individual jobs, MPI jobs, and workflows through generation of job wrapper scripts that stage, checkpoint, and execute jobs on computational resources. P-GRADE also supports monitoring of jobs and resources through tools provided by the environment, and job migration through checkpointing. Compared to P-GRADE, the GJMF provides a different approach, focusing on infrastructure for autonomic job management rather than facilitation of grid execution of applications. The GJMF assumes the existence of grid applications and provides functionality to automate the job management process, e.g., high-level abstractions for execution of groups of tasks and client APIs.

EMPEROR [63] is an OGSA-based grid meta-scheduler framework for dynamic job scheduling. EMPEROR provides a framework for performance-based scheduling optimization algorithms based on time-series analysis of job history, as well as support for advance reservations (through local resource managers). The GJMF does not perform speculative scheduling or advance reservations, but offers customization points for such mechanisms. Compared to EMPEROR, the GJMF provides a more flexible architecture, greater standardization support, and multiple levels of job management abstractions.

7. Future work

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

- **Data management.** The GJMF is envisioned to be complemented with a service-based, middleware- and transport-agnostic data management abstraction that builds on top of mechanisms such as GridFTP and Grid Storage Brokers, and integrates seamlessly with the GJMF services and service clients. Support for data management would need to be provided by implementations of GJMF middleware customization points in the JCS, as well as by GJMF service clients. Interesting research questions regarding this extension include, e.g., investigation of transport-agnostic mechanisms for integration with advanced job management mechanisms.
- **Workflow management.** While the GJMF currently integrates well with workflow management systems by offering transparent grid access, the framework itself lacks support for execution of interdependent tasks. A middleware-agnostic tool for execution of static workflows would provide a management solution similar in functionality to the higher-order services of the GJMF for tasks and task groups.
- **Adaptation of the framework to other environments.** Currently, the GJMF builds and relies on GT4 for deployment and is therefore dependent on the Apache Axis SOAP engine. The GJMF service structure is designed to keep service core functionality independent of underlying service engine, which should facilitate adaptations to alternative service environments, e.g., Apache Axis2 and Apache CXF. Preliminary investigation reveals that modification of the GJMF service security model and exposure of service state as WSRF resource properties are likely to be required for future adaptations. Adaptation of the framework to new Globus Toolkit versions is also of interest.

8. Conclusion

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

The architecture is organized in hierarchical layers of functionality, where services abstract and aggregate functionality from services in underlying layers. Services in lower layers provide explicit job submission capabilities and a fine-grained control model for the job management process while services in higher layers attempt to automate the job management process and provide a more coarse-grained control model through preconfigured job control and failure handling mechanisms. The architecture is designed to decouple grid applications from grid middlewares and infrastructure components, and abstract grid functionality through generic grid job management interfaces. Applications built on the framework are loosely coupled to underlying grids, gain portability and flexibility in deployment, and utilize heterogeneous grid resources transparently.

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

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

Acknowledgments

The authors extend their gratitude to Peter Gardfjäll, Arvid Norberg, Johan Tordsson, Mikael Öhman, Sebastian Gröhn, and Anders Häggström for prior and related work that provided a foundation for this work. This work is done in collaboration with the High Performance Computing Center North (HPC2N) and is funded by the Swedish Research Council (VR) under Contract 621-2005-3667, the Swedish National Infrastructure for Computing (SNIC), and the Swedish Government's strategic research project eSENCE.

References

- [1] E. Elmroth, P. Gardfjäll, A. Norberg, J. Tordsson, P.-O. Östberg, Designing general, composable, and middleware-independent grid infrastructure tools for multi-tiered job management, in: T. Priol, M. Vanneschi (Eds.), *Towards Next Generation Grids*, Springer-Verlag, 2007, pp. 175–184.
- [2] E. Elmroth, F. Hernández, J. Tordsson, P.-O. Östberg, Designing service-based resource management tools for a healthy grid ecosystem, in: R. Wyrzykowski, et al. (Eds.), *Parallel Processing and Applied Mathematics*, in: *Lecture Notes in Computer Science*, vol. 4967, Springer-Verlag, 2008, pp. 259–270.
- [3] E. Elmroth, P.-O. Östberg, Dynamic and transparent service compositions techniques for service-oriented grid architectures, in: S. Goriatch, P. Fragopoulou, T. Priol (Eds.), *Integrated Research in Grid Computing*, Crete University Press, 2008, pp. 323–334.
- [4] The Globus Project, An ecosystem of grid components. http://www.globus.org/grid_software/ecology.php, April 2012.
- [5] OASIS Open, Reference model for service oriented architecture 1.0. <http://www.oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf>, April 2012.
- [6] E. Elmroth, S. Holmgren, J. Lindemann, S. Toor, P.-O. Östberg, Empowering a flexible application portal with a SOA-based grid job management framework, in: A.C. Elster et al. (Eds.), *Applied Parallel Computing: State of the Art in Scientific Computing*, *Lecture Notes in Computer Science*, vol. 6127, Springer-Verlag, 2012 (in press).
- [7] The Grid Infrastructure Research & Development (GIRD) project, Umeå University, Sweden. <http://www.cs.umu.se/ds>, April 2012.
- [8] Globus. <http://www.globus.org>, April 2012.
- [9] Cluster Resources Inc. Torque resource manager. <http://www.adaptivecomputing.com/products/open-source/torque/>, April 2012.
- [10] I. Foster, A. Grimshaw, P. Lane, W. Lee, M. Morgan, S. Newhouse, S. Pickles, D. Pulsipher, C. Smith, M. Theimer, OGSA® basic execution service version 1.0. <http://www.ogf.org/documents/GFD.108.pdf>, April 2012.
- [11] D. Thain, T. Tannenbaum, M. Livny, Distributed computing in practice: the Condor experience, *Concurrency and Computation: Practice and Experience* 17 (2–4) (2005) 323–356.
- [12] I. Foster, Globus toolkit version 4: software for service-oriented systems, in: H. Jin, D. Reed, W. Jiang (Eds.), *IFIP International Conference on Network and Parallel Computing*, in: LNCS, vol. 3779, Springer-Verlag, 2005, pp. 2–13.
- [13] I. Foster, C. Kesselman, G. Tsudik, S. Tuecke, A security architecture for computational grids, in: *Proc. 5th ACM Conference on Computer and Communications Security Conference*, 1998, pp. 83–92.
- [14] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, Web Services Description Language (WSDL) 1.1. <http://www.w3.org/TR/wsdl>, April 2012.
- [15] I. Foster, J. Frey, S. Graham, S. Tuecke, K. Czajkowski, D. Ferguson, F. Leymann, M. Nally, I. Sedukhin, D. Snelling, T. Storey, W. Vambenepe, S. Weerawarana, Modeling stateful resources with web services. <http://www-106.ibm.com/developerworks/library/ws-resource/ws-modelingresources.pdf>, April 2012.
- [16] A. Anjomshoa, F. Brisard, M. Drescher, D. Fellows, A. Ly, A.S. McGough, D. Pulsipher, A. Savva, Job Submission Description Language (JSDL) specification, version 1.0. <http://www.ogf.org/documents/GFD.56.pdf>, April 2012.
- [17] I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Djaoui, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell, J. Von Reich, The open grid services architecture, version 1.5. <http://www.ogf.org/documents/GFD.80.pdf>, April 2012.
- [18] S. Graham, A. Karmarkar, J. Mischkinsky, I. Robinson and I. Sedukhin (Eds.), Web Services Resource 1.2, WS-Resource. http://docs.oasis-open.org/wsr/wsr-ws_resource-1.2-spec-os.pdf, April 2012.
- [19] S. Graham, and J. Treadwell, (Eds.), Web Services Resource Properties 1.2, WS-ResourceProperties. http://docs.oasis-open.org/wsr/wsr-ws_resource_properties-1.2-spec-os.pdf, April 2012.
- [20] S. Graham and B. Murray (Eds.), Web Services Base Notification 1.2, WS-BaseNotification, April 2012. <http://docs.oasis-open.org/wsn/2004/06/wsn-WS-BaseNotification-1.2-draft-03.pdf>.
- [21] 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* 27 (2) (2007) 219–240.
- [22] J. Frey, T. Tannenbaum, M. Livny, I. Foster, S. Tuecke, Condor-G: a computation management agent for multi-institutional grids, *Cluster Computing* 5 (3) (2002) 237–246.
- [23] P.-O. Östberg, E. Elmroth, Mediation of service overhead in service-oriented grid architectures, in: S. Jha, N. G. Felde, R. Buyya and G. Fedak, (Eds.), *Proceedings of Grid 2011: 12th IEEE/ACM International Conference on Grid Computing*, 2011, pp. 9–18.
- [24] E. Elmroth, F. Hernández, J. Tordsson, A light-weight grid workflow execution engine enabling client and middleware independence, in: R. Wyrzykowski, et al. (Eds.), *Parallel Processing and Applied Mathematics*, in: *Lecture Notes in Computer Science*, vol. 4967, Springer-Verlag, 2008, pp. 754–761.
- [25] P.-O. Östberg, A. Hellander, B. Drawert, E. Elmroth, S. Holmgren, L. Petzold, Reducing complexity in management of eScience computations, in: *Proceedings of CCGrid 2012 – The 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2012, pp. 845–852.
- [26] I. Raicu, I.T. Foster, Y. Zhao, Many-task computing for grids and supercomputers, in: *Many-Task Computing on Grids and Supercomputers*, 2008, MTAGS 2008, Workshop on, 2008, pp. 1–11.
- [27] W. Allcock (Eds.), GridFTP: protocol extensions to FTP for the grid. <http://www.ogf.org/documents/GFD.20.pdf>, April 2012.
- [28] E. Elmroth, J. Tordsson, A standards-based Grid resource brokering service supporting advance reservations, coallocation and cross-grid interoperability, *Concurrency and Computation: Practice and Experience* 25 (18) (2009) 2298–2335.
- [29] P. Gardfjäll, E. Elmroth, L. Johnsson, O. Mulmo, T. Sandholm, Scalable grid-wide capacity allocation with the SweGrid Accounting System (SGAS), *Concurrency and Computation: Practice and Experience* 20 (18) (2008) 2089–2122.

- [30] E. Elmroth, J. Tordsson, Grid resource brokering algorithms enabling advance reservations and resource selection based on performance predictions, *Future Generation Computer Systems*, The International Journal of Grid Computing: Theory, Methods and Applications 24 (6) (2008) 585–593.
- [31] S. Andreozzi, S. Burke, F. Ehm, L. Field, G. Galang, B. Konya, M. Litmaath, P. Millar, J.P. Navarro, GLUE specification v. 2.0. http://www.ogf.org/Public_Comment_Docs/Documents/2008-06/ogf-glue-2-rendering.pdf, April 2012.
- [32] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. Frystyk Nielsen, A. Karmarkar, Y. Lafon, SOAP version 1.2 part 1: messaging framework. <http://www.w3.org/TR/soap12-part1/>, April 2012.
- [33] The Globus Alliance. Globus toolkit version 4 grid security infrastructure: a standards perspective. <http://www.globus.org/toolkit/docs/4.0/security/GT4-GSI-Overview.pdf>, April 2012.
- [34] D.P. Anderson, BOINC: a system for public-resource computing and storage, in: 5th IEEE/ACM International Workshop on Grid Computing, 2004, pp. 4–10.
- [35] OMII Europe, OMII Europe—open middleware infrastructure institute, <http://omii-europe.org>, April 2012.
- [36] Grid interoperability now, <http://wiki.nesc.ac.uk/gin-jobs/>, April 2012.
- [37] N. Bobroff, L. Fong, S. Kalayci, Y. Liu, J.C. Martinez, I. Rodero, S.M. Sadjadi, D. Villegas, Enabling interoperability among meta-schedulers, in: T. Priol et al. (Eds.), CCGRID 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid, 2008, pp. 306–315.
- [38] A. Kertesz, 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, Vol. 7, Springer, 2007, pp. 25–26.
- [39] G. Pierantonio, B. Coghan, E. Kenny, O. Lyttleton, D. O'Callaghan, 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.
- [40] S. Venugopal, K. Nadiminti, H. Gibbins, R. Buyya, Designing a resource broker for heterogeneous grids, *Software: Practice and Experience* 38 (8) (2008) 793–825.
- [41] K. Krauter, R. Buyya, M. Maheswaran, A taxonomy and survey of grid resource management systems for distributed computing, *Software: Practice and Experience* 32 (2) (2002) 135–164.
- [42] E. Huedo, R.S. Montero, I.M. Llorente, A framework for adaptive execution on grids, *Software: Practice and Experience* 34 (7) (2004) 631–651.
- [43] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, M. Wilde, Falcon: a fast and light-weight task execution framework, in: Proceedings of IEEE/ACM Supercomputing 07, 2007.
- [44] H.H. Karlsen, B. Vinter, Minimum intrusion grid—the simple model, in: 14th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise, WETICE'05, 2005, pp. 305–310.
- [45] N. Furmento, W. Lee, A. Mayer, S. Newhouse, J. Darlington, ICENI: an open grid service architecture implemented with Jini, in: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, IEEE Computer Society Press, Los Alamitos, CA, USA, 2002, pp. 1–10.
- [46] L. Young, S. McGough, S. Newhouse, J. Darlington, Scheduling architecture and algorithms within the ICENI grid middleware, in: Simon Cox (Ed.), Proceedings of the UK e-Science All Hands Meeting, 2003, pp. 5–12.
- [47] I. Rodero, J. Corbalán, R.M. Badia, J. Labarta, eNANOS grid resource broker, in: P.M.A. Sloat, A.G. Hoekstra, T. Priol, A. Reinefeld, M. Bubak (Eds.), Advances in Grid Computing—EGC 2005, in: LNCS, vol. 3470, 2005, pp. 111–121.
- [48] W. Xiaohui, D. Zhaohui, Y. Shutao, H. Chang, L. Huizhen, CSF4: a WSRF compliant meta-scheduler, in: The 2006 World Congress in Computer Science, Computer Engineering, and Applied Computing, GCA'06, 2006, pp. 61–67.
- [49] G. Allen, K. Davis, T. Goodale, A. Hutano, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schott, E. Seidel, B. Ullmer, The Grid Application Toolkit: toward generic and easy application programming interfaces for the grid, *Proceedings of the IEEE* 93 (3) (2005) 534–550.
- [50] 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, I. Taylor, Enabling applications on the grid—a GridLab overview, *Int. J. High Perf. Comput. Appl.* 17 (4) (2003).
- [51] W. Lee, A.S. McGough, J. Darlington, Performance evaluation of the GridSAM job submission and monitoring system, in: UK e-Science All Hands Meeting, 2005, pp. 915–922.
- [52] M. Welsh, D. Culler, E. Brewer, SEDA: an architecture for well-connected scalable Internet services, *Operating Systems Review* 35 (5) (2001) 230–243.
- [53] R. Buyya, D. Abramson, J. Giddy, Nimrod/g: an architecture of a resource management and scheduling system in a global computational grid, 2000. CoRR, cs.DC/0009021.
- [54] 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.
- [55] R. Buyya, D. Abramson, J. Giddy, An economy driven resource management architecture for global computational power grids, 2000.
- [56] S. Venugopal, R. Buyya, L. Winton, A grid service broker for scheduling e-science applications on global data grids, *Concurrency and Computation: Practice and Experience* 18 (6) (2006) 685–699.
- [57] J.M. Alonso, V. Hernández, G. Moltó, GMarte: grid middleware to abstract remote task execution, *Concurrency and Computation: Practice and Experience* 18 (15) (2006) 2021–2036.
- [58] G. Moltó, V. Hernández, J.M. Alonso, A service-oriented WSRF-based architecture for metascheduling on computational grids, *Future Generation Computer Systems* 24 (4) (2008) 317–328.
- [59] A. Kertesz, P. Kacsuk, GMBS: a new middleware service for making grids interoperable, *Future Generation Computer Systems* 26 (4) (2010) 542–553.
- [60] T. Fahringer, R. Prodan, R. Duan, J. Hofer, F. Nadeem, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, M. Wiczorek, ASKALON: a development and grid computing environment for scientific workflows, in: I. Taylor, et al. (Eds.), *Workflows for e-Science*, Springer-Verlag, 2007, pp. 450–471.
- [61] 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, D.S. Katz, Pegasus: a framework for mapping complex scientific workflows onto distributed systems, *Scientific Programming* 13 (3) (2005) 219–237.
- [62] P. Kacsuk, G. Dozsa, J. Kovacs, R. Lovas, N. Podhorszki, Z. Balaton, G. Gombas, P-GRADE: a grid programming environment, *Journal of Grid Computing* 1 (2) (2003) 171–197.
- [63] 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.



Per-Olov Östberg is a Postdoc at the Department of Computing Science and an advanced consultant at the High Performance Computing Center North, Umeå University, Sweden. He has backgrounds in image analysis and distributed computing and holds advanced degrees (including a Ph.D.) in Computing Science from Umeå University. His research area is eScience and distributed computing with focus on virtual infrastructure and software design for distributed eScience applications. Experiences include more than half a decade of a postgraduate work in industry; six months as a visiting Postdoc at Uppsala University, Sweden, and Karolinska Institutet, Sweden; as well as a semester as a visiting scholar at ACS, Lawrence Berkeley National Laboratory, University of California, Berkeley, USA.



Erik Elmroth is Professor and Head of the Department of Computing Science at Umeå University, Sweden. His background in eScience includes Grid computing, parallel computing, algorithms for managing memory hierarchies, linear algebra library software, and ill-posed eigenvalue problems. He received the Nordea Scientific Prize 2011 and was co-recipient of the SIAM Linear Algebra Prize 2000. He currently leads the research on distributed systems at Umeå University, focusing on infrastructure and application tools for Grid and Cloud computing. International experiences include a year at NERSC, Lawrence Berkeley National Laboratory, University of California, Berkeley, USA, and one semester at the Massachusetts Institute of Technology (MIT), Cambridge, MA, USA. Erik is vice-chair of the Swedish National Infrastructure for Computing (SNIC) and he has been member of the Swedish Research Council's Committee for Research Infrastructures (KFI) as well as Chairman of its expert group on eScience. He has been appointed by the Nordic Council of Ministers (NCM) to develop Nordic and Nordic-Baltic eScience strategies.