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

Erik Elmroth and Per-Olov Östberg

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

Abstract

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

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

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

1. Introduction

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

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

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

The research question of how to best design Grid infrastructure software is currently open-ended and addressed in a number of different ways. The approach taken in this work consists of identification of a set of desirable traits likely to promote software sustainability in a Grid ecosystem (as described in Section 2), and exploration of software design and development methodologies which result in composable software components that inhabit and define niches in an ecosystem of Grid infrastructure components.

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

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

The architecture is illustrated by a proof-of-concept implementation called the *Grid Job Management Framework (GJMF)*, which is presented along with an evaluation of system performance. The framework is designed to provide transparent Grid access to applications through a set of abstractive interfaces that can be combined with (optional) advanced customizability features. Applications built on top of the framework are provided transparent Grid middleware functionality that allows Grid resource utilization without coupling applications to specific middlewares. Applications and infrastructure tools are also able to reuse components of the framework for generic Grid operation on an individual basis, promoting a functionality-based model of software reuse and increasing component and system sustainability. Throughout the paper, intended system behavior and implications of system design and architecture are discussed alongside documentation of experiences from system design and development.

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

2. The Grid Ecosystem and Software Requirements

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

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

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

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

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

3. Framework Architecture

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

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

3.1. Architecture Layers

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

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

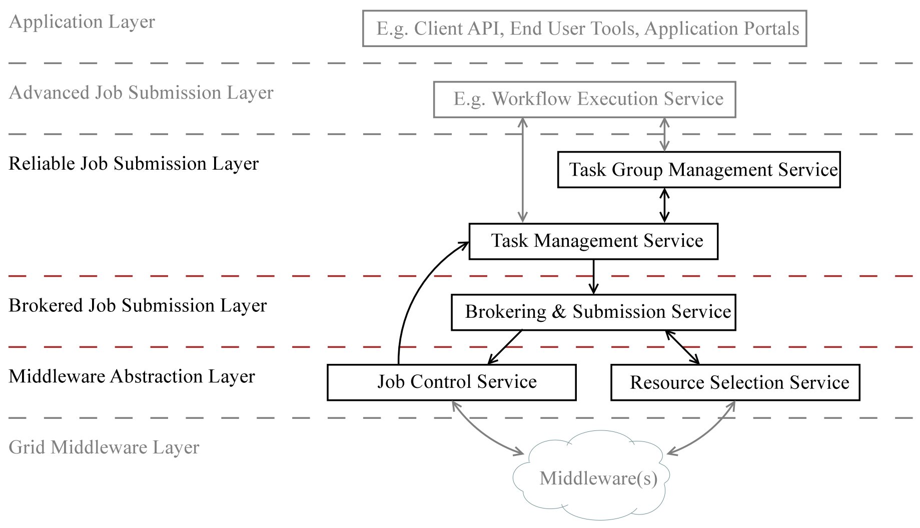

Application Layer
E.g. Client API, End User Tools, Application Portals
Advanced Job Submission Layer
E.g. Workflow Execution Service
Reliable Job Submission Layer
Task Group Management Service
Task Management Service
Brokered Job Submission Layer
Brokering & Submission Service
Middleware Abstraction Layer
Job Control Service
Resource Selection Service
Grid Middleware Layer
Middleware(s)

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

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

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

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

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

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

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

Advanced Job Submission Layer. The advanced job submission layer is in the architecture of the framework aimed towards more advanced mechanisms for job management, e.g., workflow tools, Grid application components, and portal interfaces that by functionality requirements are coupled to individual components of the framework. The services of the advanced job submission layer are intended to utilize the services of the reliable job submission layer, and function as integration bridges and customized service interfaces to the framework. Services in the advanced job submission layer are expected to provide their own job management and monitoring contexts as they are intended to aggregate the functionality of the other layers of the framework. A number of functionality sets for advanced job management have been identified and are under consideration (see Section 8) for development in the prototype implementation of the framework, e.g., management of data and sets of interdependent jobs.

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

4. The Grid Job Management Framework (GJMF)

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

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

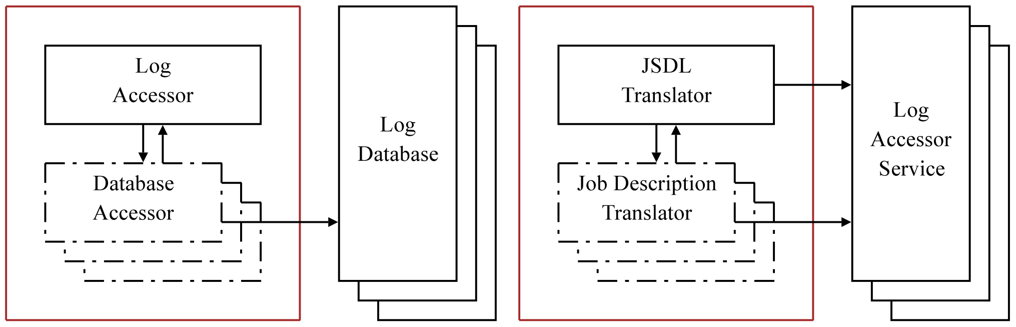
4.1. Job Definitions

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

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

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

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



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

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

4.2. Components

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

4.2.1. Log Accessor Service (LAS)

In a distributed architecture managing multiple synchronized states, ability to track state development and review processing progression is highly desirable. The Log Accessor Service (LAS) is a service that provides database-like interfaces to job, task, and task group logs generated by the GJMF. As the name suggests, the LAS is designed to provide convenient log access to services, end-users, and clients. Within the GJMF, the LAS is used to record state transitions as well as job submission and processing information. The LAS is typically expected to have monodirectional data transfers, e.g., the GJMF services use the LAS to store data, and service clients use it to inspect details of task processing.

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

4.2.2. JSDL Translation Service (JTS)

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

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

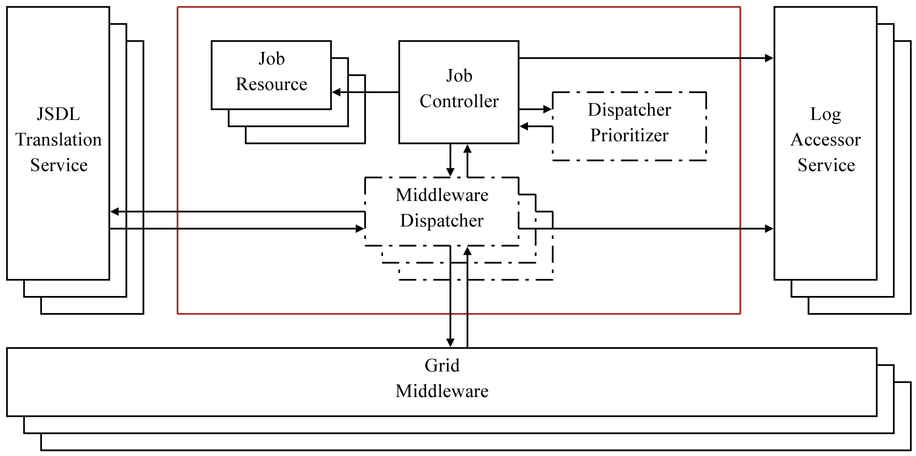


Figure 3: Internal structure of the Job Control Service. Customization points are illustrated using dotted lines.

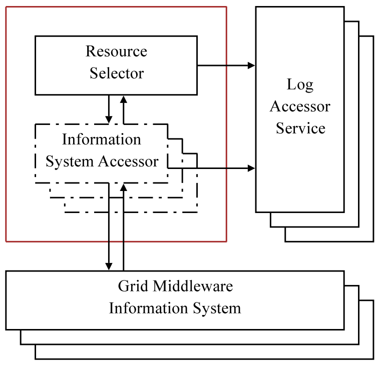
interface and abstracts the use of job description translator plug-ins. Both the JSDL translator and the translator plug-ins make optional use of LASs for log storage. Currently, the JTS supports translation between JSDL [9] and Globus Toolkit 4 Resource Specification Language (GT4 RSL) [26], NorduGrid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

4.2.3. Job Control Service (JCS)

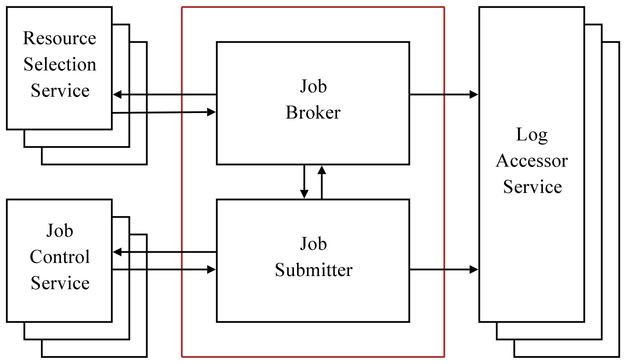
Being one of the two fundamental middleware abstraction services of the GJMF, the purpose of the Job Control Service (JCS) is to provide a uniform and middleware-independent interface for job submission and control. The JCS defines a set of generic job functionality, as well as a job state model (illustrated in Figure 8c), that provide a fundamental view of job management that other services in the GJMF build upon. Within the GJMF, the JCS is used by the Brokering & Submission Service as a job submission interface, and by the Task Management Service as a job monitoring and control interface, but the service may also be used directly by service clients as a targeted Grid job submission and control tool.

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

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



(a) The Resource Selection Service.



(b) The Brokering and Submission Service.

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

4.2.4. Resource Selection Service (RSS)

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

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

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

4.2.5. Brokering & Submission Service (BSS)

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

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

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

4.2.6. Task Management Service (TMS)

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

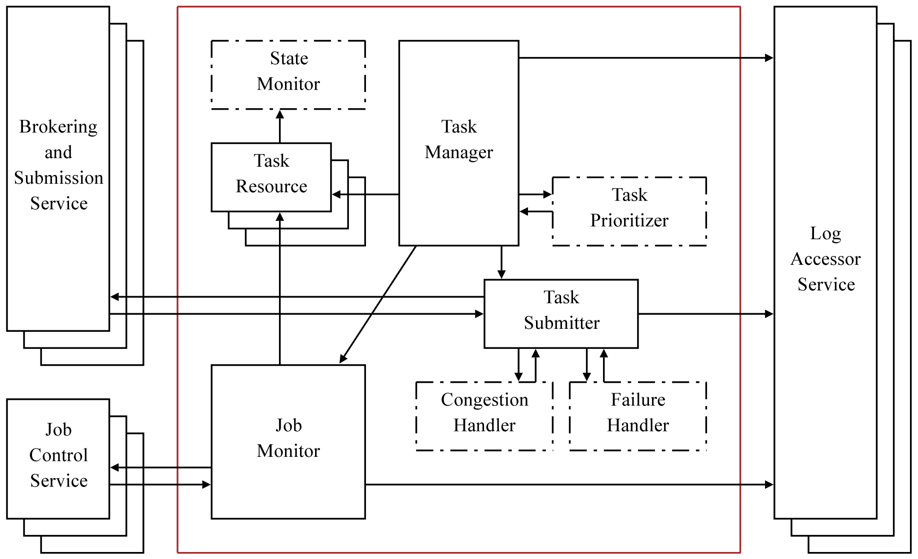


Figure 5: Internal structure of the Task Management Service. Customization points are illustrated using dotted lines.

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

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

The internal mechanisms of the TMS can be customized via configuration

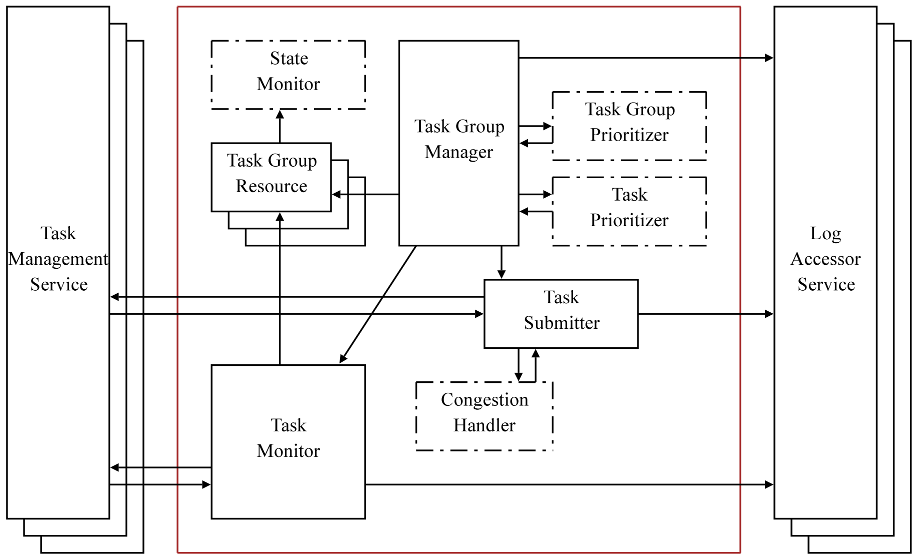


Figure 6: Internal structure of the Task Group Management Service. Customization points are illustrated using dotted lines.

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

4.2.7. Task Group Management Service (TGMS)

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

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

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

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

4.2.8. The GJMF Common Library

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

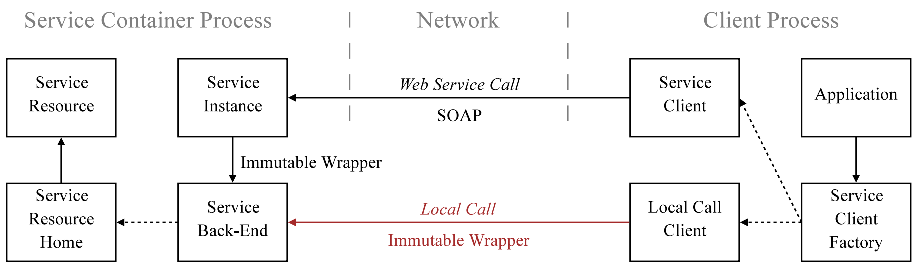


Figure 7: The GJMF service structure. The GJMF common library provides boiler-plate solutions for service instantiation, service back-end implementation, resource management, and client APIs. The GJMF client API abstracts use of the service invocation optimizations through use of service client factories.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

serialization.

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

The service development framework, in concert with the GJMF client API, handles common tasks such as data type marshalling, service instantiation, notification subscription management, and notification delivery. The framework also encapsulates a local call optimization mechanism that allows service components to be exposed as local objects to other services codeployed in the same service container, which allows co-hosted services to make marshalled in-process Java calls directly between service clients and service back-ends. This optimization mechanism, which is discussed in Section 5.1, evaluated in Section 6.3.5, and also addressed in [22], is hidden by the service structure of the common library and made completely transparent to service clients through the client API. As described in [22], the common library provides a set of basic and immutable types for use in the GJMF client API as well as a type marshalling mechanism that abstracts the use of stub types in the GJMF.

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

The GJMF common library includes four parts:

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

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

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

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

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

4.2.9. The GJMF Client Application Programming Interface

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

5. Architecture Discussion

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

5.1. Invocation Patterns

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

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

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

5.2. Deployment Scenarios

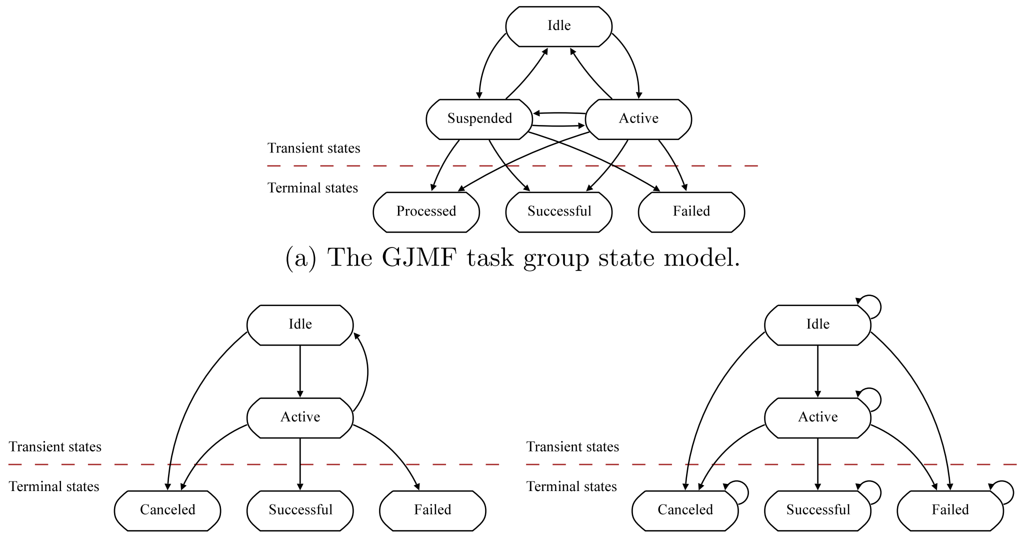
The GJMF has been designed to be as versatile as possible in terms of deployment and usage without imposing complexity of administration or loss of user control. The dynamic configuration structures, and the customizable code modules used throughout the framework provide options for modification of framework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

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

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

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

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



(a) The GJMF task group state model.

(b) The GJMF task state model.

(c) The GJMF job state model.

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

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

5.3. State Models

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

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

Table 1: GJMF state interpretations.

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

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

5.4. Data Management

To maintain middleware transparency, the GJMF does not by default actively participate in data transfers between clients and computational resources. The GJMF assumes that data files are available and can be transfered to and from computational resources by the enacting Grid middleware via a file transfer mechanism chosen by the middleware. File staging information is conveyed from clients to middlewares by the GJMF as part of job descriptions, typically in the form of GridFTP [16] URL tags in job and task JSDL.

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

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

5.5. Resource Brokering

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

In the GJMF model, the RSS has been limited to provide computational resource recommendations and rankings, services and clients are expected to handle submission and failure handling for jobs without providing feedback to the RSS. This abstraction implies that the RSS is agnostic of whether a particular execution plan is enacted or not. To compensate for middleware information system update latencies, it would be possible for the RSS to maintain an internal cache of prior execution plans and update resource load weights through speculation based on this information. As the RSS enforces user-level isolation of service capabilities, a unique cache would be created for each user and restricted to contain only recommendations for that user.

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

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

5.6. Security

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

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

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

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

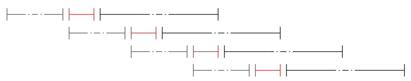
To facilitate the construction of job submission proxies, a requirement in, e.g., Grid portal construction [21], it is possible to submit a task to the GJMF specifying different credentials for job execution than those used in submission to the GJMF. For these situations, the authenticity of caller credentials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

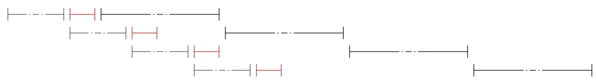
6. Performance Evaluation

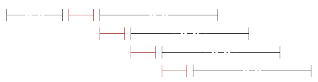
To evaluate and analyze the performance of the framework prototype we run a series of tests using a standard setup of the framework on a deployment of a Grid middleware representative of production use.

6.1. Performance Measurement

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


(a) Sequential invocation mode, infinite computational nodes.


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


(c) Batch invocation mode, infinite computational nodes.

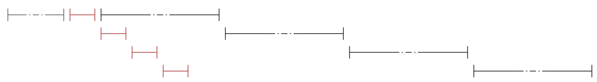

(d) Batch invocation mode, limited computational nodes. Job executions mask GJMF overhead.

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

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

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

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

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

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

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

6.2. Test Environment

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

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

We employ GT4 WS-GRAM as Grid middleware and run */bin/true* executions for ideal jobs (zero execution time) and */bin/sleep* executions for jobs with known, non-zero execution times. To maximize the impact of the GJMF overhead when testing ideal jobs, we utilize the GT4 Fork mechanism for job dispatchment. For tests of more realistic scenarios we use */bin/sleep* to get exact job execution times and use the GT4 PBS module for job dispatchment, which submits jobs to a local cluster using Torque. To minimize impact of stochastic network behaviors in our overhead measurements we do not use jobs that involve file transfers.

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

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

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

6.3. Performance Tests

The purpose of the performance evaluation is to investigate and quantify individual contributions to total system overhead, and to verify that overhead imposed by the GJMF is sufficiently small in relation to the functionality offered by the framework. In the performance evaluation, we perform a set of tests of service invocation capabilities, job submission performance, and job throughput to quantify and evaluate the impact of the GJMF overhead on the total system overhead. The tests performed are based on the overhead model presented in Section 6.1 and are designed to illustrate individual aspects of the framework overhead. The five types of tests performed are:

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

6.3.1. Job Submission

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

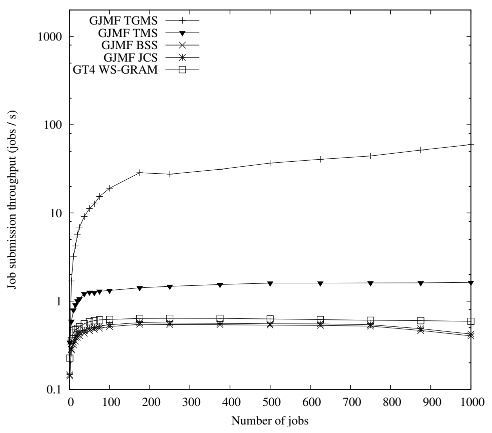
For all tests, job, task, and task group submission performance is measured as turn-around time for submission in service clients using realistic job descriptions. The average job submission makespan is used as a direct measurement of the overhead incurred by the GJMF for job submission.

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

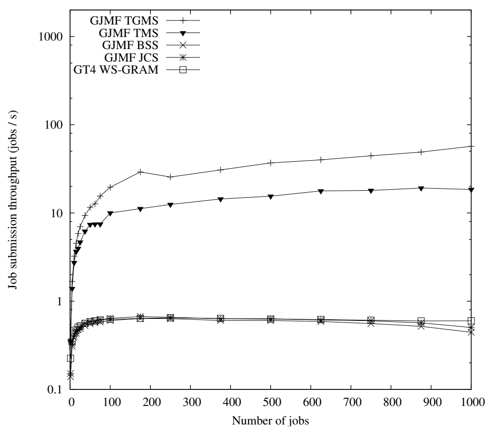
When using local call optimizations, as illustrated in Figure 10c and 10d, submission overhead can be reduced for all GJMF services. The TGMS and TMS achieve very high submission throughput due to their ability to perform asynchronous job submissions. Use of local call optimizations reduce invocation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

6.3.2. Job Throughput for Ideal Computational Settings

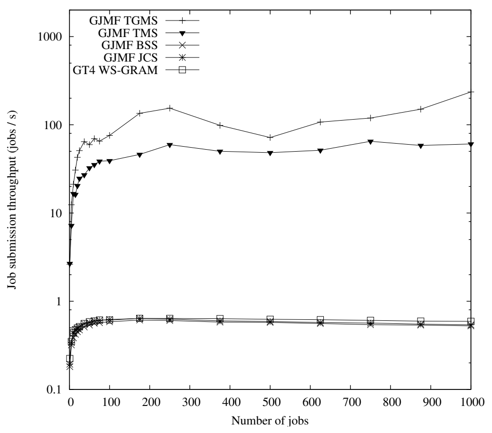
To evaluate and get an upper bound for the processing overhead component of the total system overhead, we measure the job processing capacity of the framework in terms of throughput and quantify overhead incurred by the GJMF against a baseline measurement of the GT4 WS-GRAM job processing performance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



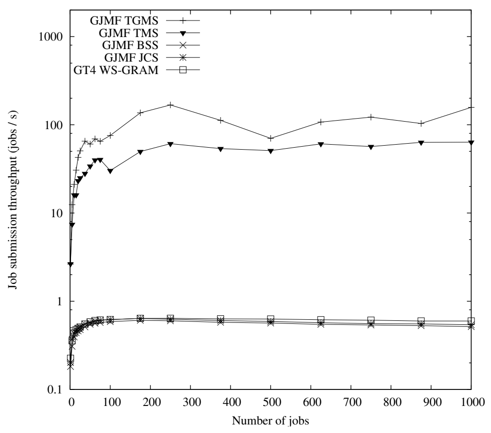
(a) GRAM and GJMF job submission throughput. Web Service invocations, sequential job submission.



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



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



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

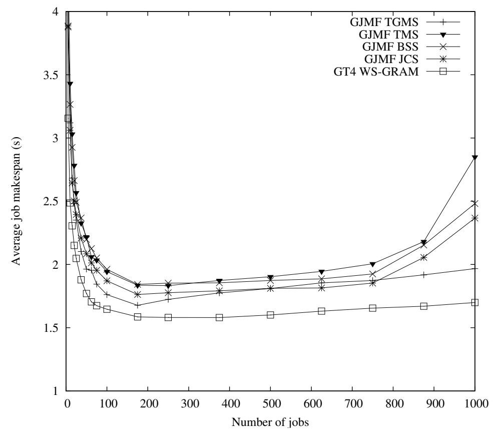
Figure 10: GRAM and GJMF job submission performance. Job submission throughput as a function of number of jobs, vertical axis logarithmic.

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

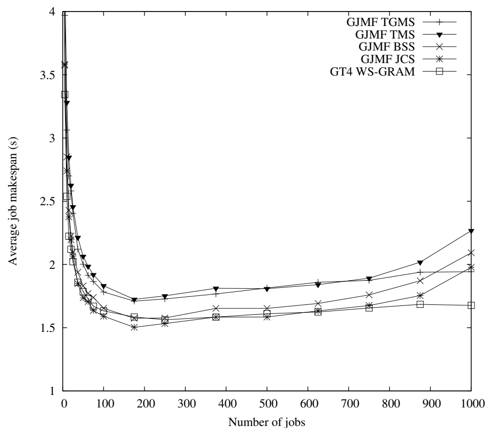
As can be seen in Figure 11, the GJMF incurs an average performance penalty of less than one second per job for ideal (zero execution time) jobs. This overhead includes factors such as job submission, interservice communication, job brokering, and distributed state management. When using batch invocation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

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

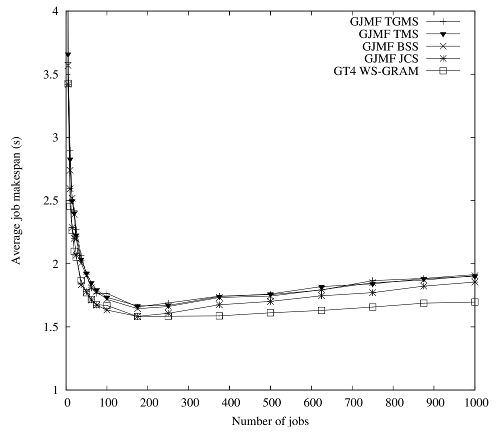
Use of the GT4 Fork mechanism for job dispatchment results in all jobs executing as spawned processes on the local machine. Despite the use of a computationally cheap process, this still causes increased load on the machine that in the measurements show as a slight decrease in average job throughput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



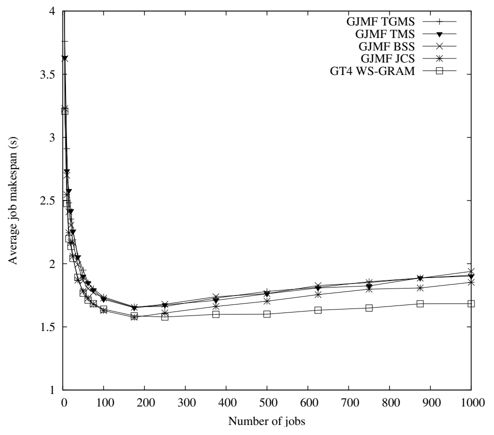
(a) GRAM and GJMF job processing throughput. Web Service invocations, sequential job submission, ideal jobs.



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



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



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

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

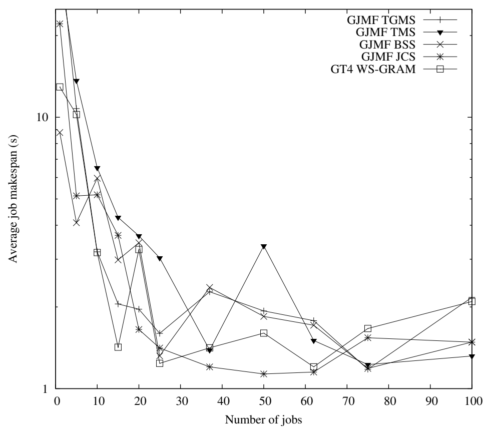
JCS in figures 11a and 11b. Note that the TGMS does not suffer from this effect as it performs single service invocations for task group submissions, and uses delays between subsequent TMS task submissions. Note also that use of batch invocation modes alleviates this effect, but does not eliminate it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

6.3.3. Job Throughput for Realistic Computational Settings

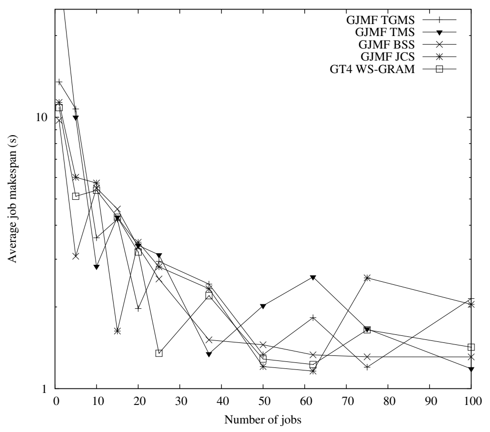
To evaluate the processing overhead component of the total system overhead under more realistic circumstances, we measure the job processing capacity of the framework in terms of throughput and quantify overhead incurred by the GJMF when deployed with a production environment system (GT4 and PBS Torque)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

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

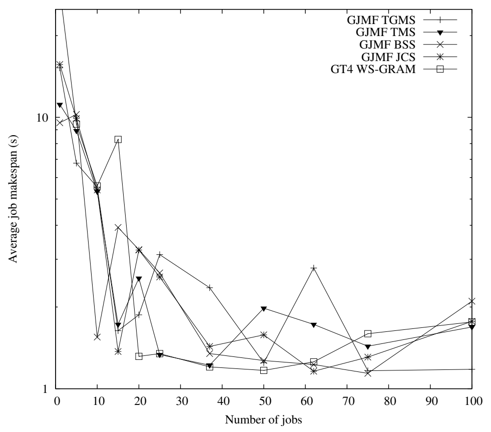
The term realistic computational settings used in this test refers to the



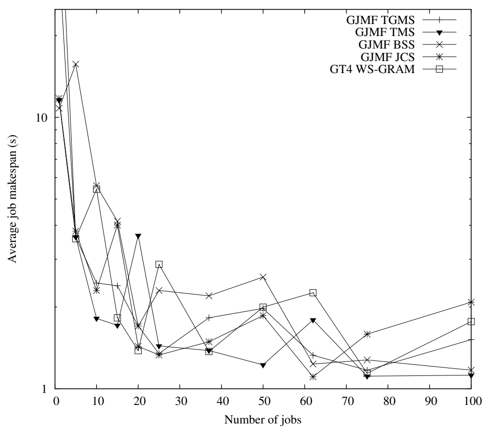
(a) GRAM and GJMF job processing throughput. Web Service invocations, sequential job submission, non-ideal jobs.



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



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



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

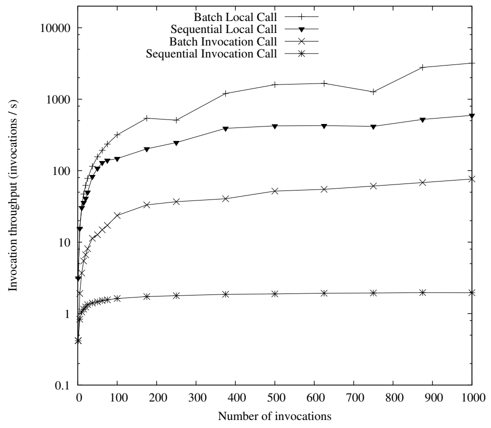
Figure 12: GRAM and GJMF job processing performance for realistic computational settings. Average job processing makespan as a function of number of jobs, vertical axis logarithmic.

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

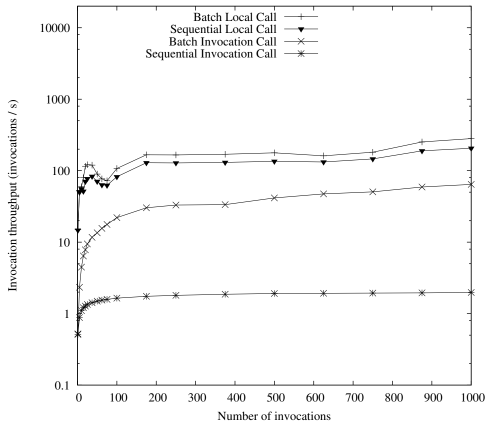
6.3.4. GJMF Auxiliary Services

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

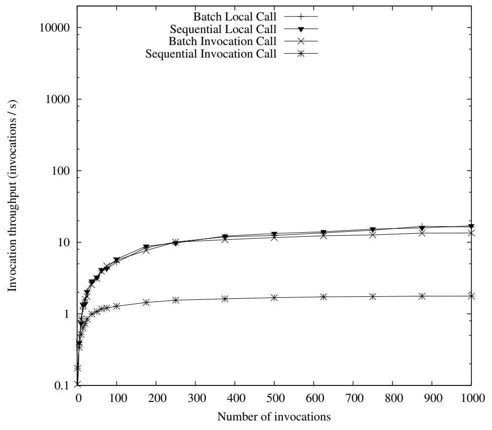
As can be seen in Figure 13, local call optimizations allow for much greater invocation throughput than Web Service invocations. This is natural as they avoid network transport and mitigate the need for message serialization and parsing. Use of batch invocation modes also increase invocation throughput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



(a) The Log Accessor Service.



(b) The JSDL Translation Service.



(c) The Resource Selection Service.

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

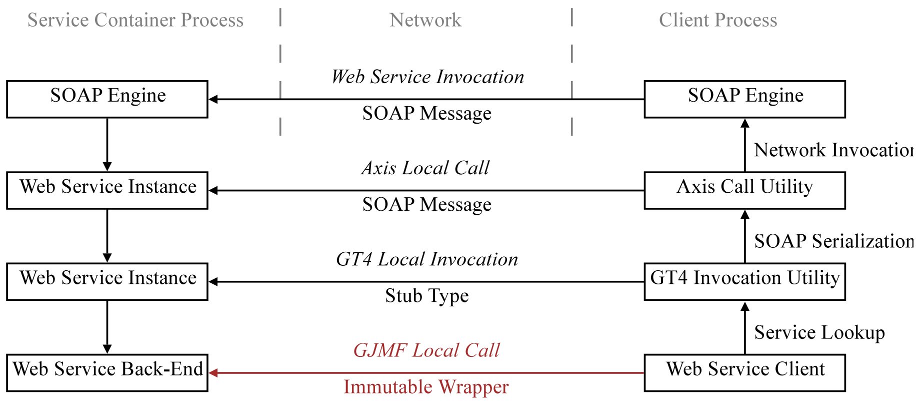


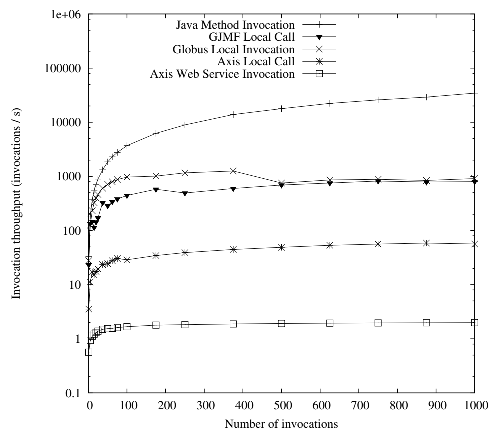
Figure 14: Local call optimization types. Illustrates actors and overhead involved.

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

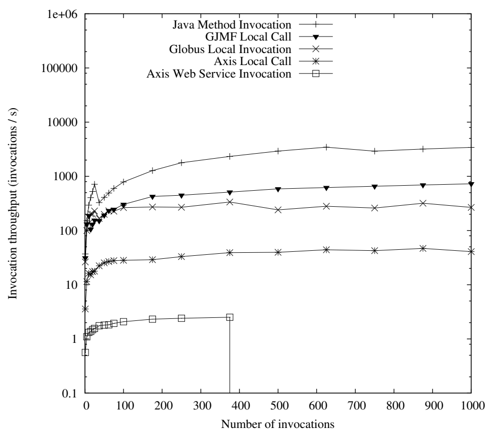
6.3.5. Local Call Optimizations

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

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



(a) Sequential invocations.



(b) Parallel invocations.

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

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

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

While not illustrated by the tests, the GJMF local call optimizations require less memory than Axis and Globus invocation optimizations as the GJMF mechanisms do not perform message serialization, maintain message contexts, or invoke message handlers for local service invocations. While this trait does not directly affect service invocation times, it reduces the memory load of the service container when using WS-BaseNotification based notification schemes for services handling large numbers of objects, e.g., a TGMS containing large task groups. As can be seen in Figure 15a, the Globus local invocation mechanism outperforms the GJMF local calls for sequential service invocations due to two factors. First, the Globus local invocation mechanism performs a caching of Web Service objects between invocations, something the GJMF is unable to do as the GJMF enforces a user-level isolation of service capabilities for each call. Second, the GJMF performs context-based type validation of job description data in immutable wrapper types, e.g., parses job descriptions and verifies that that all required information is present, a process that simplifies service development but imposes additional computational overhead. For the parallel invocation case illustrated in Figure 15b, the GJMF local call optimizations outperforms the Globus local invocations mechanism, which is attributed to the lower memory usage of the GJMF local call optimizations.

6.4. Performance Discussion

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

Furthermore, as the greater bulk of the GJMF job processing overhead is constituted of service invocation times, co-hosting the services of the GJMF allow GJMF local call optimizations to reduce the overhead contribution of the GJMF job processing mechanisms to a level where the initial job submission overhead component becomes dominant. When using standalone clients, i.e. clients not co-located with the GJMF, job submission overhead can be mediated to a one-time cost by using batch invocation modes (illustrated in 9c and 9d). In most cases, submission overhead will impact total overhead regardless of whether the GJMF is used or not, but the various invocation and deployment modes of the GJMF can be used to mediate this component. GJMF processing overhead can be mediated by GJMF deployment and configuration options, e.g., by use of codeployment of services and local call optimizations. Use of batch invocation modes for service invocations conserve network bandwidth and reduce the memory footprint of both the GJMF services and GJMF service clients. Use of local call optimizations eliminates network bandwidth requirements and reduces memory used for service invocations to a minimum.

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

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

7. Related Work

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

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

A set of job management systems exhibiting similarities in design or intended use have also been identified, and include, e.g., the GridWay Metascheduler [41], a framework for adaptive scheduling and execution of Grid jobs. Like the GJMF, GridWay builds on the Globus Tookit and offers an abstracted ("submit and forget") type of Grid job submission focused on reliable and autonomous execution of jobs. Both systems provide failover capabilities through resubmission of jobs, where GridWay offers job migration capabilities through checkpointing and migration interfaces, whereas the GJMF focuses on abstraction of Grid middleware capabilities and system composability, and offers coarse-grained resubmission policies in higher services. GridWay also offers a performance degradation mechanism which may be used to detect and trigger job migration mechanisms. The GJMF assumes computational hosts maintain acceptably consistent performance levels and relies on Grid applications and middlewares to handle checkpointing and application preemption issues.

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

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

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

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

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

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

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

The GridLab Grid Application Toolkit (GAT) [4] is a high-level application programming toolkit for Grid application development. The fundamental ideas behind the GAT and the GJMF are similar, both projects aim to decouple Grid applications from Grid middlewares by providing middleware-independent Grid access through client APIs aimed at simplifying Grid application development. The GAT builds on the GridLab [3] architecture which aims to be a complete Grid utilization platform, providing, e.g., data management services (including data transfer and replica management capabilities), monitoring services, and services for visualization of data, while the GJMF provides a composable and lean architecture for Grid utilization focusing on functionality required for job management, and relying on underlying middlewares for functionality such as job control and file staging.

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

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

The Gridbus [66] broker is a Grid broker that mediates access to distributed data and computational resources, and brokers jobs to resources based on data transfer optimality criteria. Gridbus extends the resource broker model of Nimrod-G, defining a hierarchical model for job brokering containing separate resource discovery, Grid scheduling, and monitoring components. Like in the GJMF, tasks are defined as sequences of commands that describe user requirements, including, e.g., file staging and job execution information, located within the task description itself. Task requirements drive resource discovery and tasks are resolved into jobs, here defined as units of work sent to Grid nodes, i.e. instantiations of tasks with unique combinations of parameter values. The Gridbus broker also abstracts use of multiple middlewares through a service-based interface. Differences between the two platforms include, e.g., Gridbus heuristics-based scheduling strategies, and the GJMF's ability to dynamically reconfigure framework deployment during runtime.

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

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

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

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

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

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

8. Future Work

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

- Data management. In a future extension, the GJMF is envisioned to be complemented with a service-based, middleware- and transport-independent data management abstraction that builds on top of mechanisms such as GridFTP and Grid Storage Brokers, and integrates seamlessly with the GJMF services and service clients. Support for data management would need to be provided by implementations of GJMF middleware customization points in the JCS, as well as by GJMF service clients. Interesting research questions regarding this extension include investigation of how transport-independence can be maintained while providing efficient functionality abstractions well adjusted to seamless integration with generic job management solutions.

- Workflow management. While the GJMF currently integrates with workflow management solutions by offering middleware-independent Grid job management interfaces, the framework itself lacks support for execution of interdependent tasks. Inclusion of a middleware-independent tool for execution of task graphs and static workflows would provide clients with a fire-and-forget type of workflow management solution similar to the functionality offered by the higher-order services of the GJMF for tasks and task groups.
- Evaluation of experiences from production use. Experiences from future production use of the framework is expected to provide feedback and suggest alterations or redesigns of parts of the framework.

9. Conclusion

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

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

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

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

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

10. Acknowledgements

We extend gratitude to Peter Gardfjäll, Arvid Norberg, and Johan Tordsson who's prior work and feedback have provided a foundation to build upon in this work. We acknowledge the Swedish Research Council (VR) who have supported the project under contract 621-2005-3667, and the High Performance Computer Center North (HPC2N) on who's resources the research has been performed.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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