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

Per-Olov Östberg and Erik Elmroth
Department of Computing Science, Umeå University, SE-901 87, Umeå, Sweden

Keywords: Service-oriented architecture, Grid computing, Cloud computing, Service-oriented component model.

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

1 INTRODUCTION

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

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

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

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

2 RELATED AND PRIOR WORK

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

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

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

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

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

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

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

3 GRID AND CLOUD SOFTWARE DEVELOPMENT

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

Cloud computing evolved from a number of distributed system efforts and inherits technology and methodology from Grid computing. To applications, Clouds provide the illusion of endless resource capacity through interface abstraction, and run virtual machines on resources employing hardware-enabled virtualization technologies. As the notion of a service (an always-on, network-accessible software) fits well in the Cloud computing model, many Cloud environments build on, or provide, service-oriented interfaces. In effect, Grids provide scalability through federation of resources, while Clouds provide computational elasticity through abstraction of resources.

In service-based software development, a number of trade-offs and development issues exist.

Software Reuse. Development of Grid and Cloud computing infrastructure components and applications consists, at least in part, of integration of existing systems. Integration projects tend to produce software specific to integration environments, and limit software reuse to component level. To enable software reuse, components should be kept flexible and customizable (Elmroth and Östberg, 2008; Elmroth et al., 2008), and SOA programming models should emphasize construction of modules that developers can customize without source code modification (Curbera et al., 2005).

Software Flexibility. In SOA environments, component interactions are specified in terms of service interfaces, orchestrations, and compositions. Services define interfaces in terms of service descriptions (SOAP style Web Services), or via exposure of resources through HTTP mechanisms (REST services). As SOAs are typically designed to abstract underlying execution environments and dynamically provision functionality, services may be deployed in dynamic and heterogeneous environments. To facilitate integration between components, SOAP Web Service platforms provide Application Programming Interfaces (APIs) and employ code generation tools to provide boilerplate solutions for component interaction. REST architectures define resource representations in documentation and often encapsulate component invocation in APIs. By providing mechanisms for dynamic recomposition of architectures and reconfiguration of components, SOAs facilitate system deployment and administration flexibility.

Multi-dimensional Scalability. There are many types of scalability in Grid and Cloud Computing. Within performance, scalability can be categorized in dimensions such as horizontal or vertical scalability, i.e. ability to efficiently utilize many or large resources, or in time, e.g., in terms of computation throughput, makespan, and response time. In Clouds, hardware virtualization enabled resource elasticity describes system ability to vary number and size of hardware resources contributing to Cloud resource and infrastructure capacity.

While achieving performance scalability is central to computational systems (and the explicit focus of many Grid and Cloud Computing efforts), there are also other types of scalability likely to impact software sustainability in Grid and Cloud Computing environments. Scalability in, e.g., development, deployment, and administration, are becoming limiting factors as software projects scale up. In development, scalability is often limited by system complexity and problem topology. Deployment flexibility can be measured in terms of adaptability and integrability, and is typically limited by restrictions imposed by architecture design or implementation choices. Increasingly, as software projects grow in size and complexity, configuration and administration scalability is becoming a factor. Administration scalability can be measured in terms of automation, configuration complexity, and monitoring transparency. Computational and storage scalability are often limited by problem topology, while development and deployment scalability tend to be limited by solution complexity.

Development Complexity. Limitations in current service engines, frameworks, and development tools often result in increased component implementation complexity and reduced developer productivity (when compared to non-service-based software development). Service development APIs expose low level functionality and service engine integration logic leaving, e.g., parts of message serialization tasks to service and client developers. For services defined with explicit service interface descriptions, e.g., Web Service Description Language (WSDL) documents, a number of code generation tools exist. These typically extract type systems and service interface information from service interface descriptions, and generate code to integrate and communicate with services.

Current service APIs and code generators tend to be service platform specific and tie service and service client implementations to specific service engines. Tool complexity often leads to complex interactions with and within service implementations, resulting in service interface implementations being mixed with service functionality logic. In addition, service description and type validation languages are often complex. WSDL and XML Schema are examples of widely used languages with great expressive power and steep learning curves that lower developer productivity and obstruct component reuse.

Complexity and ambiguity in service description formats lead to steep learning curves, high development overhead, and incompatibilities in service interface and message validation implementations. As generated code is intended for machine interpretation, it is typically left undocumented, unindented, and hard to read, reducing toolset transparency. Tool complexity results in vendor lock-in, reduced development productivity, and decreased software stability.

4 DESIGN METHODOLOGY

To address service software reuse, flexibility, and scalability issues, we propose a SOA development methodology consisting of two parts: an architecture design methodology and a service component model. The methodology emphasizes modularity and customization on both architecture and component level. Architectures isolate functionality blocks in services and define architectures as loosely coupled networks of services that can be customized through recomposition mechanisms. Services separate component modules and offer customization through exposure of plug-in customization points. To support this methodology we provide a service development toolset (presented in Section 5) designed to abstract service development complexity. The overall goal of this methodology is to raise service development abstraction levels and produce systems that are flexible in development, deployment, and administration.

4.1 Design Perspective

To design modular and reusable software, we employ a model of software evolution based on the notion of an ecosystem of infrastructure and application components (Elmroth et al., 2008). Here systems form niches of functionality and components are selected for use based on current client or application needs. Over time, software are subject to evolution based on natural selection. In conjunction with this model, we observe the line of reasoning used in autonomic computing (Kephart and Chess, 2003), and address scalability through modularity and reduction of software complexity. The proposed methodology provides component and architecture models well suited for construction of software self-* mechanisms.

In architecture design, we combine the top-down perspective of structured system design with the modeling of objects and relationships between objects of object-oriented programming. Similar to the reasoning of (Lau and Wang, 2007), we design components that expose functionality through well-defined Web Service interfaces and compose architectures as SOAs. System composition takes place in the component design (through interface, dependency, and communication design) and deployment (through runtime configuration) phases, and is determined through a system de- and recomposition approach (Elmroth and Östberg, 2008). As encapsulation (modularity) counters software complexity (Lau and Wang, 2007), we utilize interface abstraction and late binding techniques to construct loosely coupled and location transparent components.

The design approach defines architectures as flexible, dynamically reconfigurable, and loosely coupled networks of services. Autonomous blocks of functionality are identified and isolated, and components are modeled to keep component interactions coarse-grained and infrequent. Functionality likely to be of interest to other components or clients is exposed as services or customization points.

For applications, this approach provides flexibility in utilization and customization of system functionality, and increases system task parallelization potential. Construction of software as SOAs emphasizes composition of new systems from existing components, allows a model of software reuse where applications dynamically select components based on current needs, and facilitates replacement and updates of individual components. On component level, this approach results in increased modularity and a greater focus on interface abstraction, benefiting component and system flexibility, adaptability, and longevity.

While architectures designed as networks of services may be distributed, components are likely to (at least partially) be co-hosted for reasons of performance and ease of administration. Co-hosted components are able to make use of local call optimizations, which greatly reduce service container memory and CPU load, as well as system network bandwidth requirements. Use of local call optimizations results in less network congestion issues, fewer network stack package drops, fewer container message queue drops, and reduced impact of component communication overhead and errors. Local call optimization mechanisms allows design of systems that combine the component communication efficiency of monolithic systems with the deployment flexibility of distributed service-based systems.

4.2 Architecture Design

Our design approach is summarized in three steps.

Identification of Autonomous Functionality Blocks. Similar to how objects and object relationships are modeled in object-oriented programming, autonomous functionality blocks are identified and block interactions are modeled using coarse-grained service communication patterns. Key to this approach is to strike a balance between architecture fragmentation and the need to keep components small, single-purpose, and intuitive. Component dependency patterns are identified to illuminate opportunities for parallelization of system tasks.

Identification of Exposure Mechanisms for Functionality Blocks. Functionality of interest to components in neighboring ecosystem niches, with clear levels of abstraction, and where well-defined interfaces can be defined is exposed as services. Functionality not of interest to other systems, but where component customization would increase system flexibility is exposed as customization points, e.g., through configuration and plug-ins. Service-exposed functionality is generally identified at architecture level, while customization points are typically identified at component level.

Design of Service Interactions and Interfaces. Formulation of interfaces and service communication patterns are essential to performance efficiency in SOAs. Key to our approach is to design architectures that allow services to function efficiently as both local objects and distributed services. As exposure of components as services may lead to unexpected invocation patterns, defensive programming techniques and design patterns are employed to keep service interfaces unambiguous, simple, and lean.

4.3 Component Design

In component design, we adhere to the general principles for a service-oriented component model presented in (Cervantes and Hall, 2004) and organize service components in a way similar to classic three-tier architectures. To enable a software development model facilitating loose coupling of service components, we emphasize separation of modules in component design (Yang and Papazoglou, 2004). Separation of service client, interface, logic, and storage components facilitates flexibility in distributed system development and integration. Separation of service clients and interface implementations is a key mechanism in Service-Oriented Computing (SOC) that facilitates loose coupling in systems, and is here extended to provide (optional) flexibility in logic implementation.

Use of language and platform independent techniques for data marshaling and transportation allows service clients to be implemented using application-specific languages and tools, facilitating system integrability and adoptability. Development of service components using this pattern can be used to, e.g., create lightweight Web Service integration interfaces for existing components running in component environments such as the Common Object Model (COM) or Enterprise Java Beans (EJB).

Separation of service interface and logic implementation enables use of alternative wire protocols and communication paradigms, and facilitates implementation and deployment of service logic in component model environments. Encapsulation of platform-specific code, i.e. service client and interface implementations, facilitates migration and porting of service logic to alternative service platforms.

Implementation of local call optimizations allow logic components to function as local Java objects in service clients (Elmroth and Östberg, 2008), reducing component communication overhead to the levels of monolithic architectures (Östberg and Elmroth, 2011). Embedding local call optimizations in component APIs allows optimizations to be transparent to developers and ubiquitous in service implementations (see Section 5), combining the deployment flexibility of distributed SOAs with the communication performance of monolithic architectures.

Implementation of service logic in component models introduce additional requirements for software development. Best practices for Web Service and component-based software development overlap partially. Web Service components should, e.g., be implemented to be thread safe, communicate asynchronously, consume minimal system resources, and minimize service response times. Separation of service logic from storage layers enables loose coupling between component models and storage mechanisms, and facilitates migration of service logic between component environments.

To provide component-level flexibility, we define structures for customization points as plug-ins. Component interfaces are defined for advisory and functionality provider interfaces, and customization point implementations are dynamically loaded at runtime. Through this mechanism, third parties can replace, update, and provide plug-in components for internal structures inside services without impacting design of application architectures.

5 DEVELOPMENT TOOLSET

Software reuse in highly specialized, complex, and low maturity environments such as emerging Grid and Cloud computing eScience platforms is limited and inefficient. Code generation tools target automation of software development and can facilitate software reuse by providing boiler-plate solutions for service communication and isolate service logic. To address software reuse and abstraction of development complexity, we present a service development toolset called the Service Development Abstraction Toolset (SDAT). SDAT builds on the component de-

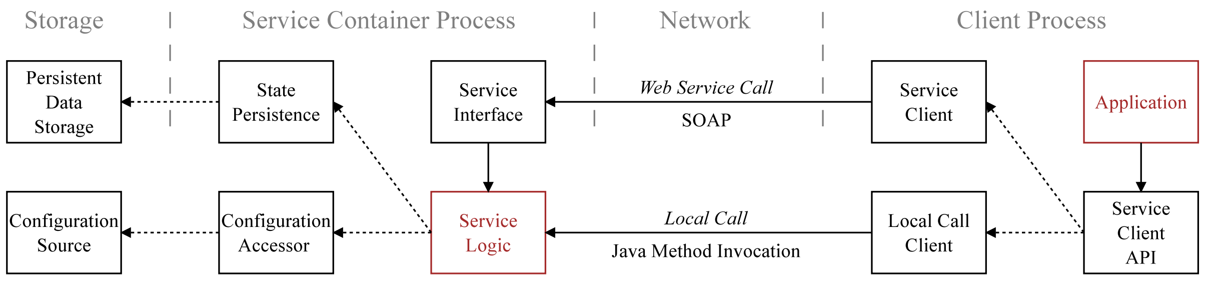

Figure 1: SDAT service structure. Manually developed components (application and service logic) decoupled from generated components (service invocation, configuration management, and state persistence). Transparent local call optimizations reduce service invocation overhead and container load.

sign model of Section 4.3 and is designed to raise service development abstraction levels through a simplified service description language and a code generation tool that separates service components and provides boiler-plate solutions for, e.g., data representation, validation, and communication.

5.1 Simplified Service Description

To promote loose coupling of service clients and implementations, enable service code generation, and facilitate service discovery, we define a simplistic XML-based service interface description format called the Abstractive Service Description Language (ASDL). The format specifies service interfaces in terms of tree-based data types and call by value operations on defined types. Data types are defined in a schema language defined as a subset of XML Schema. Conceptually, ASDL can be seen as a minimalistic subset of the Web Service Description Language (WSDL), where the expressive power of XML Schema and WSDL are reduced in favor of simplicity in interface interpretation and data representation.

The ASDL type schema language consists of a restricted set of XML and XML Schema tags. Data field types are defined using `simpleType` tags, and organized in hierarchical records using `complexType` tags. For message validation schema completeness `element` tags are inserted, and service interfaces are defined using `service` and `operation` tags. All schemas define a target namespace, use explicitly referenced namespaces, and all tags are qualified. XML Schema mechanisms for `include` and `import` tags are supported to facilitate type definition reuse. ASDL is designed to provide a service interface design model semantically equivalent to Java interfaces using immutable Java classes.

Through ASDL, SDAT exploits XML Schema document validation without encumbering interface designers with the full complexity of XML Schema and WSDL. For translation to WSDL interface descriptions and generation of SOAP Web Service implementations, the following assumptions are made. All service communication is kept document-oriented and use literal encoding of messages. Interfaces are designed to have single-part messages and define a single service per service description. Services define a single type set per service description and all data fields are encoded in XML elements. Exceptions are exposed as SOAP faults and serialized as messages defined in the service type schema.

5.2 Code Generation

To facilitate design of architectures as loosely coupled networks of services, SDAT defines a service structure (illustrated in Figure 1) that isolates service functionality blocks and employs a local call optimization mechanism for co-hosted services. Local call optimizations transparently bypass network serialization and reduce invocation overhead and container load (Elmroth and Östberg, 2008). Immutable data types are exposed in service interfaces and used for service invocation. Optional message validation is performed in service client and interface components. The service structure defines modules for:

- Data type representations. Flat record structures are extracted from service description schemas and data type representation components are defined as immutable and serializable Java classes.
- Service interfaces. A service invocation framework abstracting call optimizations is built into service implementations and client APIs, making optimizations transparent to service clients.
- Message data validation. XML Schemas are extracted from service interface descriptions and used to generate message validation components.
- Service configuration management. A configuration accessor and monitor API is defined for service components. Configuration modules are available to service client and logic components.

- Persistent state storage. A framework for persistent service state storage is defined and accessible to service logic components. Persistence modules are customizable and can be extended to support, e.g., additional databases or serialization formats.

Separation of service interface and logic implementation allows compartmentalization of service platform specific code and facilitates abstraction of service interface and invocation implementation.

As illustrated in Figure 1, the SDAT service structure isolates manually coded components (application and service logic) from generated service components. Typical service development using SDAT consists of specification of a ASDL interface, generation of service components and interfaces, and implementation of a service logic interface. The goal of the tool is to abstract service development complexity to the level of implementation of a Java interface while providing optional customization of service components.

In addition to service components, SDAT generates deployment information (WSDL service descriptions, deployment descriptors, etc.), security code (WS-Security implementations, policy files, etc.), and a (Apache Ant) build environment. By default, SDAT generates compiled and deployable service packages where developers only need to add service logic implementations to services. To facilitate transparency, all source code generated is designed to be well formatted, easy to read, and documented.

Configuration of SDAT services is segmented into separate container and service configuration. As SDAT services expose customization points in the form of plug-ins for, e.g., service logic implementation, some configuration of the generated SDAT framework must be done on container level. A typical example of this is service plug-ins, which are specified in container Java runtime property settings. Configuration of service logic components is typically done through service configuration files accessed through the SDAT configuration API.

To enforce user-level isolation of service capabilities, service interface implementations instantiate unique service logic components for each invoking user. User-level isolation is implemented through a singleton factory that caches service instances based on invocation credentials, i.e. user certificates. This mechanism is designed to coexist with native mechanisms for component-level isolation of services.

For platform independence, clear representations of service interfaces, strong Web Service support, and in-memory compilation, SDAT is built and produces services in Java. For separation of service interface specification and service development, SDAT primarily supports SOAP style Web Services. Currently, SDAT integrates with the Apache Axis2 SOAP engine, but as service interface components are decoupled from service logic components, support for additional service engines, client languages, or communication patterns can be extended without affecting other service mechanisms. In extension, this model is expected to be of interest for creating components that can be hosted in different service engines, or even (simultaneously) support multiple types of service communication (e.g., REST, TCP/IP, and SOAP).

The aim of SDAT is to provide a development tool that abstracts complex and error-prone service communication development and allows service developers to focus on service logic. The tool is designed to provide a simplistic service model where service interfaces are kept simple, but still have expressive power enough to create efficiently communicating services. SDAT is designed as a prototype development tool that aims to integrate with existing service containers and development environments. While current code generators are tied to particular service environments or languages, SDAT is designed to support a development methodology rather than a specific platform or toolset.

6 CASE STUDIES

To illustrate our design approach, we present case studies from two recent software development projects. In these projects, which target development of Grid infrastructure components, emphasis is placed on development of flexible architectures capable of seamless integration into existing Grid and High-Performance Computing (HPC) environments.

6.1 GJMF

The Grid Job Management Framework (GJMF) (Östberg and Elmroth, 2010) illustrated in Figure 2 is a Grid infrastructure component built as a loosely coupled network of services. Designed to provide middleware-agnostic and abstractive job management interfaces, the GJMF offers concurrent access to multiple Grid middlewares through components organized in hierarchical layers. Services in higher layers aggregate functionality of lower layers, and form a rich middleware-agnostic Grid job management interface. Through a set of integration plug-ins, the framework can be customized to, e.g., support additional Grid middlewares, replace job brokering algorithms, and define job failure management policies. Framework composition, plug-in selection, and component configuration are configurable

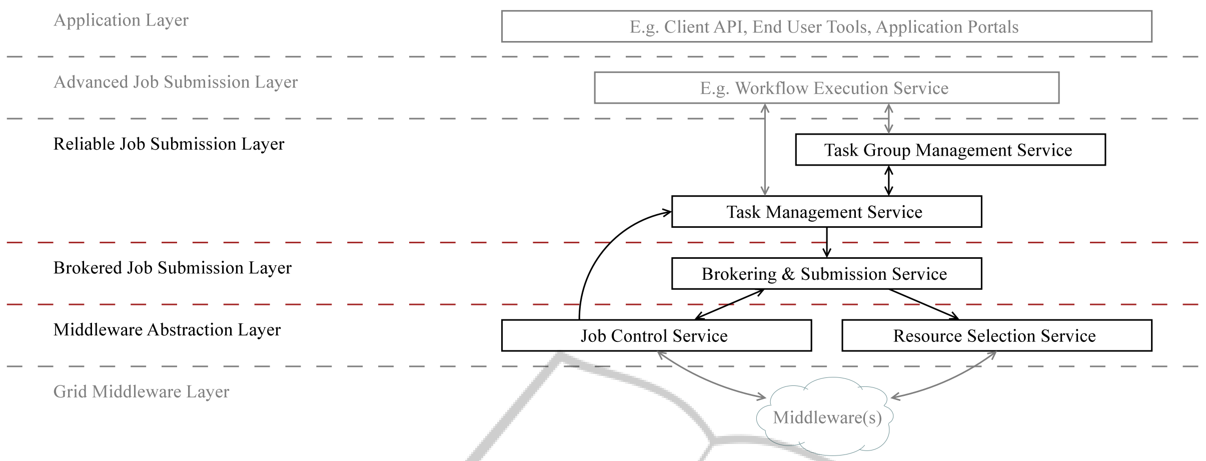
Application Layer
E.g. Client API, End User Tools, Application Portals
Advanced Job Submission Layer
E.g. Workflow Execution Service
Reliable Job Submission Layer
Task Group Management Service
Task Management Service
Brokered Job Submission Layer
Brokering & Submission Service
Middleware Abstraction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

Figure 2: The Grid Job Management Framework (GJMF). A hierarchical framework of services offering abstractive Grid job management. Illustration from (Östberg and Elmroth, 2010).

through dynamic configuration modules.

The GJMF predates the service development toolset of Section 5 and has served as a testbed for the development methodology. The framework is developed using the Globus Toolkit v4 (Foster, 2005) and employs WSRF notifications for service monitoring and coordination. Due to the flexibility of the framework architecture, the GJMF is deployable in multiple concurrent settings as, e.g., a gateway job management interface, an alternative job brokering mechanism, or a personal client-side job monitoring tool.

6.2 FSGrid

FSGrid (Östberg et al., 2011) is a job prioritization mechanisms for scheduler-based Grid fairshare policy enforcement. Designed as a distributed architecture consisting of a network of services, FSGrid is segmentable and can be tailored to resource site deployments. The system virtualizes usage metrics and resource site capacity, and is capable of collaboration between different types of FSGrid configurations.

As illustrated in Figure 3, FSGrid deploys components to be close to data and computation, and enforces VO and resource site allocation policies simultaneously. The system mounts VO allocation policies onto resource site policies, assembles and operates on global usage data, and injects a fairshare job prioritization mechanism into local scheduling decisions.

The flexibility of the architecture allows FSGrid to be deployed in different configurations on different sites, to be dynamically reconfigured, and adapt to dynamic changes in usage data and allocation policies. FSGrid exposes customization points through dynamic service configuration modules, and plug-ins for usage decay functions and fairshare metrics.

In FSGrid, all service interfaces are describe using ASDL and all service components developed using SDAT. The code generation tools of SDAT help to isolate service implementations from service container and communication dependencies. The system is deployed using Apache Axis2 (Apache Web Services Project - Axis2, 2011) service containers and integrates into cluster schedulers using custom service clients. As GJMF and FSGrid are designed using the same methodology but different tools and platforms, they make a suitable platform for evaluation.

6.3 Evaluation

The GJMF and FSGrid are developed using the same approach to SOA design, and designed with the same goal: to provide flexible architectures for Grid infrastructure. Both systems are designed as loosely coupled and reconfigurable networks of services, expose customization points for tailoring of component functionality, and provide APIs to facilitate integration into deployment environments.

Experiences from integration of GJMF with the LUNARC application portal (Lindemann and Sandberg, 2005) and a problem-solving environment in R are documented in (Elmroth et al., 2011) and (Jayawardena et al., 2010) respectively. These projects illustrate benefits of construction of infrastructure components as flexible networks of service SOAs. The GJMF exposes a range of job submission, monitoring, and control interfaces that can be utilized to integrate in heterogeneous deployment environments. The deployment flexibility of the framework allows parts of the framework to fulfill different job management roles and be hosted separately.

A recent performance analysis (Östberg and Elm-

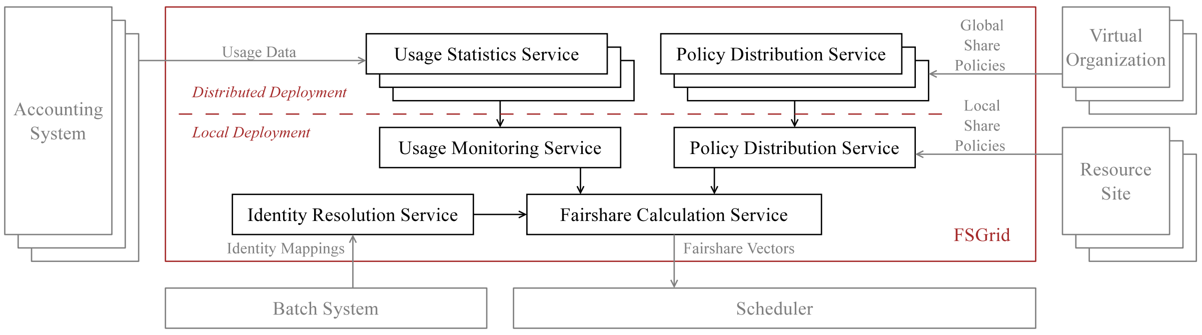

Figure 3: FSGrid, a scheduler-based fairshare job prioritization framework for Grid environments built as a distributable set of services. Illustration from (Östberg et al., 2011).

roth, 2011) illustrates that organization of services in hierarchical layers allows the GJMF to mask service communication overhead through parallel task processing. Local call optimizations significantly reduce communication overhead and container load for inter-service communication. The service structure abstracts use of local call optimizations and allows optimizations to be ubiquitous and transparent.

Compared to GJMF development, the FSGrid project benefits from use of SDAT in reduction of service interface implementation complexity. FSGrid development cycles are shorter, and use of ASDL and SDAT facilitates experimentation in architecture design. FSGrid benefits from use of SDAT in compartmentalization and reduction of complexity in service logic implementation. The proposed development methodology provides GJMF and FSGrid architecture and component level reconfigurability, adaptivity, and flexibility. The SDAT abstracts service development and facilitates porting of core system functionality to alternative service platforms.

Impact of the proposed methodology on the internal quality of produced software can be evaluated through, e.g., evaluation of the maintainability and cohesion of service interfaces (Perepletchikov et al., 2010). Study of the impact of SDAT on the quality of GJMF and FSGrid is subject for future work.

7 CONCLUSIONS

In this paper we address software development practices for eScience applications and infrastructure. We discuss service-based software development in Grid and Cloud computing environments and identify a set of current software development issues, e.g., complexity and lack of flexibility in service development tools. To address these issues, we propose a SOA-based software design methodology constituted by a set of architecture design guidelines, a component design model, and a toolset designed to abstract service development complexity. The approach is illustrated and evaluated in a case study of experiences from two recent software development projects.

Our design approach aims to produce software architectures that are flexible in deployment, reduce need for complex distributed state synchronization, and facilitate distribution and parallelization of system tasks. The component model isolates service components in standalone modules, exposes functionality as services and customizable plug-ins, and compartmentalizes platform-specific service interface and invocation code. The toolset is designed to support the design methodology through abstraction of development complexity and facilitation of flexibility in service development, deployment, and utilization. A simplified service description language abstracts service interface and type description complexity.

Experiences from recent software development projects illustrate the need for structured development models for Service-Oriented Architectures. The hierarchical structure of the GJMF allows the framework to dynamically function as several types of job management interfaces simultaneously as well as mask service invocation overhead. Building the system as a network of services allows FSGrid to deploy components close to data and computations as well as provide more flexible interfaces for scheduler integration. The design approach provides both systems with increased flexibility in development and deployment. Use of the SDAT toolset abstracts service development complexity and provides FSGrid a component model that supports dynamic reconfiguration and encapsulation of platform-specific service functionality.

ACKNOWLEDGEMENTS

The authors extend their gratitude to the anonymous reviewers, Deb Agarwal, and Vladimir Vlassov for valuable feedback that has contributed to the quality of this paper. This work is funded in part by the Swedish Research Council (VR) under Contract 621-2005-3667, the Swedish Government's strategic research project eSSENCE, and the European Community's Seventh Framework Programme (FP7/2001-2013) under grant agreement 257115 (OPTIMIS). The authors acknowledge the Lawrence Berkeley National Laboratory (LBNL) for supporting the project under U.S. Department of Energy Contract DE-AC02-05CH11231.

REFERENCES

Apache Web Services Project - Axis2 (2011). http://ws.apache.org/axis2, February 2011.

Arsanjani, A., Ghosh, S., Allam, A., Abdollah, T., Ganapathy, S., and Holley, K. (2010). SOMA: A method for developing service-oriented solutions. *IBM Systems Journal*, 47(3):377–396.

Cervantes, H. and Hall, R. (2004). Autonomous adaptation to dynamic availability using a service-oriented component model. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 614 – 623.

Curbera, F., Ferguson, D., Nally, M., and Stockton, M. (2005). Toward a programming model for service-oriented computing. In Benatallah, B., Casati, F., and Traverso, P., editors, *Service-Oriented Computing - ICSOC 2005*, volume 3826 of *Lecture Notes in Computer Science*, pages 33–47. Springer Berlin / Heidelberg.

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

Elmroth, E., Holmgren, S., Lindemann, J., Toor, S., and Östberg, P.-O. (to appear, 2011). Empowering a Flexible Application Portal with a SOA-based Grid Job Management Framework. In *The 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing*.

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

Escoffier, C., Hall, R. S., and Lalanda, P. (2007). iPOJO: an Extensible Service-Oriented Component Framework. *Services Computing, IEEE International Conference on*, 0:474–481.

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

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

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

Lau, K.-K., Velasco Elizondo, P., and Wang, Z. (2005). Exogenous connectors for software components. In Heineman, G. T., Crnkovic, I., Schmidt, H. W., Stafford, J. A., Szyperski, C., and Wallnau, K., editors, *Component-Based Software Engineering*, volume 3489 of *Lecture Notes in Computer Science*, pages 90–106. Springer Berlin / Heidelberg.

Lau, K.-K. and Wang, Z. (2007). Software component models. *Software Engineering, IEEE Transactions on*, 33(10):709 –724.

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

Lowy, J. (2005). *Programming .NET Components, 2nd Edition*. O'Reilly Media, Inc.

OSGi (2011). http://www.osgi.org, February 2011.

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

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

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

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

Perepletchikov, M., Ryan, C., and Tari, Z. (2010). The impact of service cohesion on the analyzability of service-oriented software. *IEEE Transactions on Services Computing*, 3:89–103.

Spring Framework (2011). http://www.springsource.org, February 2011.

Yang, J. and Papazoglou, M. P. (2004). Service components for managing the life-cycle of service compositions. *Information Systems*, 29(2):97 – 125. The 14th International Conference on Advanced Information Systems Engineering (CAiSE*02).