DYNAMIC AND TRANSPARENT SERVICE COMPOSITION TECHNIQUES FOR SERVICE-ORIENTED GRID ARCHITECTURES*

Erik Elmroth and Per-Olov Östberg
Dept. Computing Science and HPC2N, Umeå University, SE-901 87 Umeå, Sweden
{elmroth, p-o}@cs.umu.se
http://www.gird.se

Abstract With the introduction of the Service-Oriented Architecture design paradigm, service composition has become a central methodology for developing Grid software. We present an approach to Grid software development consisting of architectural design patterns for service de-composition and service re-composition. The patterns presented can each be used individually, but provide synergistic effects when combined as described in a unified framework. Software design patterns are employed to provide structure in design for service-based software development. Service APIs and immutable data wrappers are used to simplify service client development and isolate service clients from details of underlying service engine architectures. The use of local call structures greatly reduces inter-service communication overhead for co-located services, and service API factories are used to make local calls transparent to service client developers. Light-weight and dynamically replaceable plug-ins provide structure for decision support and integration points. A dynamic configuration scheme provides coordination of service efforts and synchronization of service interactions in a user-centric manner. When using local calls and dynamic configuration for creating networks of cooperating services, the need for generic service monitoring solutions becomes apparent and is addressed by service monitoring interfaces. We present these techniques along with their intended use in the context of software development for service-oriented Grid architectures.

Keywords: Grid software development, Service-Oriented Architecture, Web Service composition, Design patterns, Grid ecosystem.

*Financial support has been received from The Swedish Research Council (VR) under contract number 621-2005-3667. This research was conducted using the resources of the High Performance Computing Center North (HPC2N).

1. Introduction

With the introduction of service-oriented computing and the increased popularity of the Service-Oriented Architecture (SOA) design paradigm, service composition has become a key methodology for building distributed, service-based applications. In this work we outline the foundational concepts of our SOA development methodology, introducing and describing a number of techniques targeting the development of robust, scalable, and flexible Grid software. We investigate development methodologies such as design patterns, call optimizations, plug-in structures, and techniques for dynamic service configuration. When combined, these techniques make up the foundation of an approach for composable Web Services that are to be used in Grid SOA environments. The techniques are here presented in Grid Web Service development scenarios.

The outline of the paper is the following: A motivation and overview of our work is presented in Section 2. A more detailed introduction to the concept and aspects of service composition is given in Section 3, after which we present architectural design patterns used to address these concepts in Section 4. Finally, a brief survey of related work is presented in Section 5, followed by conclusions in Section 6 and acknowledgements.

2. Motivation and Overview

The work presented here has grown out of a need for flexible development techniques for the creation of efficient and composable Web Services. Current Grid systems employ more and more SOA-based software where scalability is a key requirement on all levels of system design, including in the development process. Service composition techniques, which employ services as building blocks in applications through the use of service aggregators, often create systems that impose substantial overhead in terms of memory requirements and execution time. Although Web Services are distributed by definition, utilizing them dynamically is often a process with lack of flexibility and transparency. The complexity of SOAP message processing alone can present impracticalities to SOA developers, as a single Web Service that exchanges large or frequent messages may in itself negatively impact the performance of other, co-located, services.

In our approach, we address these issues in two ways; by providing *flexible and transparent structures for dynamic reconfiguration of (networks of) services*, and by outlining *development patterns for optimization of interaction between co-located services and service components*. More specifically, we provide a set of architectural software design patterns for service APIs, local call structures, flexible plug-in and configuration architectures, and service monitoring facilities. Combined, these techniques make up a framework that serves to reduce the temporal and spatial system footprints (time of ex-

ecution and memory requirements, respectively) of co-located services, and provide for a software development model where dynamic service composition is made transparent to service client developers. The techniques presented are completely orthogonal to approaches using the Business Process Execution Language for Web Services (BPEL4WS) [9], Web Service Choreography Interface (WSCI) and similar techniques for service composition, and the resulting Web Services can be used in a range of service orchestration and choreography scenarios.

The approach presented here has emerged from work on the Grid Job Management Framework (GJMF) [3], a software developed in the Grid Infrastructure Research & Development (GIRD) [14] project. As a key part of the GIRD project, we investigate software development methodologies for the Grid ecosystem [13], an ecosystem of niched software components where component survival follows from evolution and natural selection [5], and a Grid built on such components. We primarily develop software in Java using the Globus Toolkit 4 (GT4) Java WS Core [7], which contains an implementation of the Web Services Resource Framework (WSRF).

3. Service Composition Techniques

Two approaches to service composition are service orchestration and service choreography. As the needs and practices in Grid and Web Service software development vary, clear definitions of the terms are yet to be fully agreed upon. In Peltz [11], service orchestration and choreography are described as approaches to create business processes from composite Web Services. Furthermore, service orchestration is detailed to be concerned with the message-level interactions of (composite and constituent) Web Services, describing business logic and goals to be realized, and representing the control flow of a single party in the message exchange. Service choreography is defined in terms of public message exchanges between multiple parties, to be more collaborative by nature, and taking a system-wide perspective of the interaction, allowing involved parties to describe their respective service interactions themselves.

Our approach to service composition is primarily concerned with transparency and scalability in dynamic service usage. We investigate techniques for developing Web Services in a dynamic and efficient manner, Web Services that can be transparently de-composed and dynamically re-composed.

3.1 Transparent Service De-Composition

At system level, Web Services are defined in terms of their interfaces without making any assumptions about the internal workings of the service functionality. In SOA design, focus is on service interactions rather than service design, and a service set providing required functionality is assumed to exist.

In the development of individual services, the structured software development approach is often hindered by the practical limitations of Web Services. By recursively subdividing the functionality of a composite Web Service, a process here referred to as service de-composition, it is often possible to identify functionality that can be reused by other services if exposed as Web Services. However, response times and memory requirements of Web Services often make it impractical to expose core functionality in this manner.

We address this issue with a framework for call optimizations, which allows software components to simultaneously and transparently function as both Web Services and local Java objects in co-located services. Small, single purpose components are easier to develop and maintain, less error-prone, and often better matched to standardized functionality [5]. By mediating the technical limitations imposed by Web Services, the use of these techniques provides a programming model that offers transparency in the use of services in distributed object-oriented modelling. As these techniques are optimizations of calls between co-located services, they are completely orthogonal to, and can be used in conjunction with, service composition techniques such as BPEL4WS, WS-AtomicTransaction and WS-Coordination.

A recent example of the application of these techniques is the construction of a workflow execution engine. A workflow engine typically contains functionality for, e.g., workflow state coordination, task submission, job monitoring, and log maintenance. By de-composing the engine functionality into a set of cooperating services rather than a large, monolithic structure, reusable software components are created and can be exposed as Web Services. The use of the proposed call optimization framework makes the de-composition process transparent to developers, provides improved fault tolerance though the use of multiple service providers (for, e.g., job submission), and preserves the performance of a single software component (an example from [3] and [4]).

3.2 Dynamic Service Re-Composition

Given a mechanism for service de-composition, a natural next step is to identify mechanisms to facilitate dynamic and transparent reconfiguration of Web Services during runtime, here referred to as service re-composition. In most service orchestration and choreography scenarios, this can be achieved using late service binding and dynamic discovery of services. As in the case of service de-composition, natural inefficiencies in these techniques may discourage developers from using them to their full potential.

We employ a scheme for dynamic configuration of services into networks of smaller, constituent services. Once again, this is a lower-level optimization of the service interactions that does not compete with traditional service orchestration techniques, but can rather co-exist with them. The scheme (out-

lined in Section 4.5) consists of services keeping local copies of configuration modules that may at any time be updated by external means. All services consult their respective configuration modules when making decisions about what plug-ins to load, which services to interact with, etc. Once a transaction with another service has been initialized, information about this process is maintained separately. The benefits of using this scheme include increased flexibility in development and deployment; access to transparent mechanisms for redundancy, fault tolerance and load balancing; and ease of administration.

A practical example of the application of this technique is the internal workings of the GJMF [3]. All services in the framework are configured using the dynamic configuration technique described, allowing services to reshape the network of services that collectively make up the higher-level functionality of the framework. Note that this technique is completely transparent to service orchestration and choreography approaches as it operates on a lower level. In fact, in a service orchestration scenario it is expected that the configuration data would be provided the service by the orchestration mechanism itself.

4. Architectural Design Patterns

The techniques presented here are intended to be used as architectural design patterns to facilitate the development of scalable and composable Web Services. Though they may be used individually, the techniques have proven to provide synergistic effects when combined, both in development and deployment.

4.1 Software Design Patterns

In architecture design, we extensively employ the use of established software design patterns [8] for the creation of efficient and reusable software components with small system footprints. The Flyweight, Builder, and Immutable patterns are used to create lean and efficient data structures. Patterns such as the Singleton, Factory Method, and Observer are deployed in a variety of scenarios to create dynamic and composable software components. To enable components to dynamically update and replace functionality, we use the Strategy, Abstract Factory, Model-View-Controller, and Chain of Responsibility patterns. The Facade, Mediator, Proxy, Command, Broker, Memento, and Adapter patterns are used to facilitate, organize, abstract, and virtualize component interaction.

4.2 Immutable Wrappers & Service APIs

In this section, we present patterns used for data representation and service APIs. The techniques presented combine design patterns and design heuristics, and are aimed to simplify service client development and facilitate the techniques presented in the following sections.

Passive data objects such as job and workflow descriptions are rarely modified once created. A useful pattern for the representation of passive data objects is to construct immutable data wrapper classes that provide abstraction of the data interface. Embedding data validation in wrappers also simplifies data handling, and is considered good practice in defensive programming. Typically, in Web Service development, data representations are specified in service descriptions and stub types are generated from WSDL. The use of wrappers around stub types provides the additional benefit of encapsulating service engine-specific stub behavior and incompatibility issues between service engines. The practice of assigning unique identifiers, e.g., in the form of Universally Unique Identifiers (UUID), to data instances facilitates the use of persistence models such as Java object serialization and GT4 resource persistence, and provides services and clients with synchronized data identifiers. By creating a service-specific data translation component, it is possible to help service instances to translate stubs to wrappers, and vice versa. The use of immutable wrappers and a designated translation component is illustrated in Figure 1. In the figure, software components are illustrated as boxes, component interactions as solid arrows, and dynamically discovered and resolved interactions as arrows with dotted lines. Note that the service client APIs and back-end make use of immutable data wrappers and are isolated from the stubs by the stub type translator.

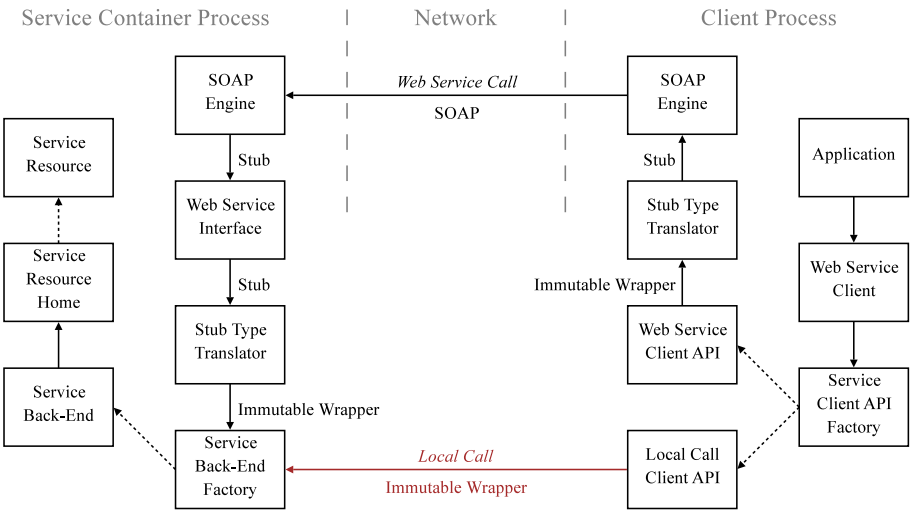

Figure 1. Illustration of local call optimizations for co-located services; dynamic resolution of service client APIs, back-ends and resources; and the use of immutable data wrappers.

In the interest of software usability for developers, it is recommended to provide client APIs with each Web Service. This practice allows developers

with limited experience of Web Service development to use SOAs transparently, and offers reference implementations detailing service use. In service APIs, a programming language interface, rather than a concrete implementation, should be used to abstract the service interface. The API interface should furthermore make strict use of wrapped data types in order to isolate it from changes in underlying architectures, e.g., Web Service engine replacement.

4.3 Local Call Structures

The use of local call structures facilitates the development of components that can be used both as generic objects and stand-alone Web Services. As illustrated in Figure 1, we propose a structure where Web Service implementations are divided into separate components for service data, interface, and implementation. Here, the service data are modeled as WSRF resources, which are dynamically resolved through the resource home using unique resource identifiers. The service interface contains the actual Web Service interfaces, and handles call semantics, stub type translation, and parameter validation issues. The service implementation back-end houses the service logic. It is dynamically resolved using a service back-end factory that instantiates a unique service implementation for each user, providing complete user-level isolation of service capabilities and resources.

Separating the service interface from the service implementation makes it possible for service clients that are co-located with the service (i.e., other services running in the same service container) to directly access the service logic. As illustrated in Figure 1, local calls bypass resource consuming data translations, credentials delegations, and Web Service invocations. For service notification invocations, the process is mediated through a notification dispatcher that dynamically resolves service resources and provides optional notification filtering and translation. Note that this scheme allows the GT4 resource persistence mechanisms to function unhindered, and remains compatible with the WSRF and WS-Notification specifications.

The resolution of the service back-end, and the local call logic, are encapsulated and made transparent to developers through the use of service client API classes. A service API factory provides appropriate service API implementations based on inspection of the service URLs, e.g., comparing IP address and port number to the local service containers configuration to determine if a local call can be made and wrapping the use of multiple (stateless) service instances into a single, logical service client interface. The service API factory makes this process transparent to the developer, which provides a set of service URLs to retrieve a service client interface.

The use of local calls efficiently optimizes communication between co-located services, but the main benefit of the technique is that it allows for

transparent de-composition of service functionality into networks of services. This provides for a more flexible development model for services that can be dynamically re-composed with a minimum of overhead, a requirement for service networks that rely on state update notifications for service coordination.

4.4 Policy Advisor and Mechanism Provider Plug-Ins

For situations where modules are to be dynamically provided and reused within components, but not between them, we make use of dynamic plug-in structures. Made up by a combination of programming language interfaces and designated configuration points, plug-in modules are dynamically located and loaded, and are considered volatile in the sense that they are intended to be short-lived and dynamically replaceable.

Functionality provided by plug-ins can be divided into two major categories: policy advisors and mechanism providers. A policy advisor implementation is intended to function in a strict advisory capacity for scenarios where policy logic is too complex to be expressed in direct configuration. The typical role of a policy advisor is to provide decisions when asked specific questions (formulated by the plug-in interface). This type of plug-in is useful for decision support in, e.g., failure handling or job prioritization. Mechanism providers are typically used for interface abstraction and integration point exposure. These types of modules are used to provide, e.g., vendor-specific database accessors or alternative brokering algorithms for job submitters.

Plug-in implementations should be light-weight, refrain from causing side-effects, have short response times, be thread-safe, and use minimal amounts of memory. Services using plug-ins should acquire the modules dynamically for each use, and rely strictly on the plug-in interface for functionality. As plug-ins can be provided by third party developers, and dynamically provided over networks, the use of code signing techniques to maintain service integrity is advisable. Grid security solutions that deploy Public-Key Infrastructures (PKI) for associating X.509 certificates with users can also be used to provide key pairs for code signing. When services provide user-centric views of service functionality, per-user configuration of service mechanism is trivial to realize.

4.5 Dynamic Service Configuration

Configuration data for Web Services are typically expressed in XML and loaded from local configuration files. Semantic Web Services provide configuration metadata to facilitate a higher degree of automation in, primarily, service composition and choreography. Similar to this approach, we employ a simplistic architecture for dynamic configuration built on the interchange of configuration data between services, and customized configuration modules to be used within services. This approach allows services to be expressed as net-

works of services, and to dynamically adapt to changes in executional context in a way that can be utilized by semantic service aggregators.

Central to our configuration approach is a dynamically replaceable configuration module. Each service maintains a configuration module factory that instantiates configuration modules when needed. The manner in which data contained in the configuration modules are acquired is encapsulated in the factory and can alternate between, e.g., polling of configuration files, triggering in databases, querying of Grid Monitoring and Discovery Services, and notifications from dedicated configuration services.

Providing configuration data through dedicated configuration services allows for transparent configuration of multiple services, where each service requests configuration data based on current user identity and service location. Dedicated configuration services can monitor resource availability and perform, e.g., load balancing through dynamic reconfiguration of networks of cooperating services. In terms of administrational overhead, this technique can alleviate the managerial burden of administrating services as it provides a single point of configuration for multiple service containers. As the local call structures of Section 4.3 provide an automatic and transparent optimization of calls between co-located services, the configuration service may attempt to optimize inter-service usage by favoring cooperation between co-located services.

In this scheme, services should never maintain direct references to configuration modules, but rather rely on them as temporary factories for configuration data. Interpretation of configuration data, type conversions, and data validation are examples of tasks to be performed by configuration modules. The use of caching techniques for configuration modules, and the synchronization and acquisition of raw configuration data should be encapsulated in configuration module factories. As seen in Section 4.4, configuration data may also be supplied in the form of plug-ins, in which case the configuration module is responsible for the location and dynamic construction of these plug-ins. When providing sensitive data, the personalization techniques of Section 4.3 can be used to provide user-level isolation of service configuration.

4.6 Service Monitoring

The dynamic configuration solutions of Section 4.5 facilitate the deployment of composite Web Services as networks of services. For reasons of system transparency, it is equally important to make parts of this configuration available to service clients, e.g., as WSRF resource properties. Consider a client submitting workflows to a workflow execution service, which schedules and submits a Grid job for each workflow task. In the interest of system openness, the client should be provided means to trace job execution, e.g., from workflow down to computational resource level. By publishing job End-Point References (EPR),

or log service URLs, the service empowers clients with the ability to monitor and trace job execution.

As mentioned in Section 4.2, data entities are provided unique identifiers prior to Web Service submission. Using these identifiers as resource keys for corresponding WSRF resources in Web Services allow clients with knowledge of identifiers (and service URL) to create resource EPRs when needed. Stateful services expose interfaces for listing resources contained in the services. For efficiency, the information returned by these interfaces are limited to lists of data identifiers (UUIDs). To improve usability and ease of development for service clients, boiler-plate solutions for tools to monitor service content are provided with each service. Although not further explored here, it should be noted that these monitoring interfaces, as well as the wrappers and service APIs of Section 4.2, are well suited for use in web portals and directly usable in the JavaService Pages (JSP) environment.

5. Related Work

There exists numerous valuable contributions on how to design for service composition and orchestration within both the fields of Grid computing and service orientation. For reasons of brevity, however, this section only references a selected number of related publications that directly touch upon the concepts presented in our software development approach.

The authors of [6] provide a grouping of service composition strategies. Our approach, containing late service bindings and semi-automatic service interaction planning, falls into the semi-dynamic service composition strategies category of this model. Brief surveys of service orchestration and choreography techniques are given in [10] and [11], and an approach for developing pattern-based service coordination is presented in [15]. Our work focuses on design heuristics and patterns for dynamic and transparent service composition in Grid contexts, and is considered orthogonal to all these techniques. The authors of [2] investigate a framework for service composition using Higher Order Components. Here, component Web Service interface generation is automated, and services are dynamically configured and deployed. We consider this a different technique pursuing a similar goal, i.e., dynamic service composition.

The Globus Toolkit [7] and the Apache Axis Web Service engine both contain utilities for local call optimizations. The Axis engine provides an in-memory call mechanism, and the Globus Toolkit provides a configurable local invocation utility that performs dynamically resolved Java calls to methods in co-located services. These approaches provide a higher level of transparency in service development, whereas our approach focuses on transparency for service client developers. In terms of performance, direct Java calls are naturally faster than in-memory Web Service invocations, and the GT4 approach suffers additional

overhead for the dynamic invocation of methods compared to our approach. Additionally, GT4 does not currently support local invocations for notifications.

Recent approaches to Grid job monitoring are presented in [1] and [12], and are here included to illustrate service monitoring functionality in dynamically composable service networks. We strive to provide dynamic monitoring and traceability mechanisms that are usable in external service monitoring tools, rather than providing stand-alone service monitoring solutions.

6. Conclusions

We present an approach to Grid software development consisting of a number of architectural design patterns. These patterns, as presented in Section 4, provide a framework addressing service de- and re-composition. The patterns presented can each be used individually, but provide synergistic effects when combined into a framework. E.g., the unique identifiers of the immutable wrappers that are used in service client APIs can also be used as resource keys for service resources, providing a simple mechanism for client-service data synchronization. Additional examples of synergistic effects are the cooperative use of local call structures, dynamic configuration, plug-ins, and service monitoring techniques: Local call structures reduce service footprints to a level where services are usable for the creation of transparent service networks. As service APIs and service API factories make the use of local calls transparent, service client developers are given an automated mechanism for optimization of service interaction. Employing dynamic configuration techniques to exploit the transparency of local calls then further increases flexibility in service interaction and administration of multiple services. Plug-ins can in turn be used to represent policy decisions, i.e., configuration semantics too complex to be represented in direct configuration, to provide alternative mechanisms, and expose integration points in services. Parts of service configuration can be exposed through monitoring interfaces to provide system transparency and monitorability, and services can employ replaceable plug-ins to utilize customized monitoring mechanisms.

The patterns described provide individually useful mechanisms for system architecture, and are orthogonal in design to each other and related technologies. Combined, they provide a framework for building lean and efficient Web Services that can be used transparently in cooperative networks of services.

Acknowledgments

We are grateful to Johan Tordsson and the anonymous referees for providing valuable feedback on, and improving the quality of, this work.

References

[1] A. N. Duarte, P. Nyczyk, A. Retico, and D. Vicinanza. Global Grid monitoring: the EGEE/WLCG case. In *GMW '07: Proceedings of the 2007 workshop on Grid monitoring*, pages 9–16, New York, NY, USA, 2007. ACM.

[2] J. Dünnweber, S. Gorlatch, F. Baude, V. Legrand, and N. Parlavantzas. Towards automatic creation of Web Services for Grid component composition. In V. Getov, editor, *Proceedings of the Workshop on Grid Systems, Tools and Environments, 12 October 2005, Sophia Antipolis, France*, December 2006.

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

[4] E. Elmroth, F. Hernández, and J. Tordsson. A light-weight Grid workflow execution engine enabling client and middleware independence. In R. Wyrzykowski et.al, editors, *Parallel Processing and Applied Mathematics. 7th Int. Conference, PPAM 2007*. Lecture Notes in Computer Science, Springer Verlag, 2007 (to appear).

[5] E. Elmroth, F. Hernández, J. Tordsson, and P-O. Östberg. Designing service-based resource management tools for a healthy Grid ecosystem. In R. Wyrzykowski et al., editors, *Parallel Processing and Applied Mathematics. 7th Int. Conference, PPAM 2007*. Lecture Notes in Computer Science, Springer-Verlag, 2007 (to appear).

[6] M. Fluegge, I. J. G. Santos, N. P. Tizzo, and E. R. M. Madeira. Challenges and techniques on the road to dynamically compose Web Services. In *ICWE '06: Proceedings of the 6th international conference on Web engineering*, pages 40–47, New York, NY, USA, 2006. ACM.

[7] I. Foster. Globus toolkit version 4: Software for service-oriented systems. In H. Jin et al., editors, *IFIP International Conference on Network and Parallel Computing,* Lecture Notes in Computer Science 3779, pages 2–13. Springer-Verlag, 2005.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[9] IBM. Business Process Execution Language for Web Services, version 1.1. http://www.ibm.com/developerworks/library/specification/ws-bpel/, visited February 2008.

[10] N. Milanovic and M. Malek. Current Solutions for Web Service Composition. *IEEE Internet Computing*, 08(6):51–59, 2004.

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

[12] M. Ruda, A. Křenek, M. Mulač, J. Pospíšil, and Z. Šustr. A uniform job monitoring service in multiple job universes. In *GMW '07: Proceedings of the 2007 workshop on Grid monitoring*, pages 17–22, New York, NY, USA, 2007. ACM.

[13] The Globus Project. An "ecosystem" of Grid components. http://www.globus.org/grid_software/ecology.php, visited February 2008.

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

[15] C. Zirpins, W. Lamersdorf, and T. Baier. Flexible coordination of service interaction patterns. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 49–56, New York, NY, USA, 2004. ACM.