

The Anatomy and Physiology of the Grid Revisited

Chris A. Mattmann^{1,2}, Joshua Garcia¹, Ivo Krka¹, Daniel Popescu¹, and Nenad Medvidovic¹

¹*Computer Science Department
University of Southern California
Los Angeles, CA 90089, USA
{joshuaga,krka,dpopescu,veno}@usc.edu*

²*Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109, USA
mattmann@jpl.nasa.gov*

Abstract

In this paper, we revisit the widely cited domain-specific software architecture (DSSA) for the domain of grid computing. We have comprehensively studied 18 grid systems over the past five years, observing that, while individual grid systems are widely used and deemed successful, the grid DSSA is underspecified and makes it difficult to pinpoint what makes a software system a grid. Inspired by this conclusion, we describe our work-in-progress towards answering this important question.

1. Introduction

Over the past half-century, computing has undergone several transformations that have fundamentally changed the manner in which humans use computers and the nature of problems that can be solved with computers. Grid computing [1, 2] is a recent advance that shows promise of enabling another such transformation. The grid allows virtually any person or organization to solve a variety of complex problems by utilizing the computing resources beyond those of just a small cluster of computers. Today, grids have been used successfully in several domains, including cancer research, planetary science, and earth science [3].

A number of technologies have emerged, claiming to be grid technologies or grid platforms (e.g., see Table 1). Our own work resulted in two such technologies. The first, OODT [3], is a data grid platform currently in use at NASA and National Cancer Institute’s Early Detection Research Network. The second, GLIDE [4], is a mobile grid platform. The early literature also resulted in several “big picture” publications that tried to establish the underlying principles of the grid: its “anatomy” [2] and “physiology” [1], which describe the grid’s software architecture, as well as its overarching requirements. Even though their authors likely did not view them that way, these reference requirements and architecture together comprised a domain-specific software architecture (DSSA) [5] for the domain of grid computing.

In our previous work, we commenced a pilot study [6] in which we attempted to extract a subset of the architectural principles underlying the grid from the grid’s published “anatomy” [2]. We then recovered, using the source code and existing documentation, the architectures of five widely used grid technologies, and compared those architectures to the anatomy. While it was difficult to draw definitive conclusions given the scope of that study, we observed a number of discrepancies that suggested that the published anatomy of the grid is not reflective of the existing grid systems.

These discrepancies served as an impetus to significantly expand the study. We partially report on our results in this paper. We first elaborated the proposed grid DSSA by revisiting the architecturally relevant aspects of the grid’s published “anatomy” [2] as well as its “physiology” [1]. We then analyzed the source code, documentation, and usage of eighteen widely deployed grid technologies, including the five from our pilot study, in order to recover their architectures and compare them to the published grid DSSA.

The paper is organized as follows. Section 2 highlights related studies in understanding grid technologies and architectural recovery. Section 3 summarizes the results of our analysis of grid technologies and concludes the paper.

2. Background and Related Work

In this section, we first discuss existing studies of the grid. We then provide an overview of the architectural recovery techniques used in this research.

2.1. Studies of the Grid

Two seminal studies that have tried to underpin and motivate grid technologies have been conducted by Kesselman and Foster, highlighting the grid’s *anatomy* [2] and *physiology* [1]. The two comprise the grid’s DSSA. In the interest of space, we will only summarize the key facets of the DSSA.

The anatomy of the grid is defined as a five-layer architecture with several overarching requirements.

Table 1. The studied grid technologies.

Technology	PL	KSLOC	#ofModules	URL
Alchemi	C# (.NET)	26.2	186	http://www.alchemi.net/
Apache Hadoop	Java, C/C++	66.5	1643	http://hadoop.apache.org/
Apache HBase	Java, Ruby, Thrift	14.1	362	http://hadoop.apache.org/hbase/
Condor	Java, C/C++	51.6	962	http://www.cs.wisc.edu/condor/
DSpace	Java	23.4	217	http://www.dspace.org/
Ganglia	C	19.3	22	http://ganglia.info/
GLIDE	Java	2	57	http://sunset.usc.edu/~softarch/GLIDE/
Globus 4.0 (GT 4.0)	Java, C/C++	2218.7	2522	http://www.globus.org/
Grid Datafarm	Java, C	51.4	220	http://datafarm.apgrid.org/
Gridbus Broker	Java	30.5	566	http://www.gridbus.org/
Jcgrid	Java	6.7	150	http://jcgrid.sourceforge.net/
ODDT	Java	14	320	http://oodt.jpl.nasa.gov/
Pegasus	Java, C	79	659	http://pegasus.isi.edu/
SciFlo	Python	18.5	129	http://sciflo.jpl.nasa.gov/
iRODS	Java, C/C++	84.1	163	https://www.irods.org/
Sun Grid Engine	Java, C/C++	265.1	572	http://gridengine.sunsource.net/
Unicore	Java	571	3665	http://www.unicore.eu/
Wings	Java	8.8	97	http://www.isi.edu/ikcap/wings/

- (1) *Application* – The top-most layer houses custom applications that plug into the common services of an underlying grid infrastructure.
- (2) *Collective* – The next layer aggregates underlying *Resource* layer services, agglomerating information for a given grid application.
- (3) *Resource* – This layer encapsulates underlying heterogeneous computing resources (such as files, disks, I/O, etc.) and provides a standard interface for communicating with grid services.
- (4) *Connectivity* – This layer is responsible for providing security, communication, and coordination of access from grid resources to underlying physical resources present in the bottom-most grid layer.
- (5) *Fabric* – The bottom-most layer’s elements include low level DBMS, disk I/O, threading and other OS-like resources, available from individual nodes in a grid.

The anatomy disallows “upcalls”, i.e., inter-layer interaction initiated by a lower layer, where a call is an aggregate of explicit invocations from one component to another. The anatomy is ambiguous as to whether it is possible to “skip” layers, i.e., whether interactions can involve non-neighboring layers; the most widely referenced diagram from [2] implies that the layers are opaque. Finally, the nature of inter-layer interactions (i.e., connectors [7]) is not elaborated; all interactions are treated as direct (local or remote) procedure calls.

The physiology of the grid focuses on the *Resource* layer of the reference architecture. It defines the core requirements and canonical services for grid resources, calling this work the “Open Grid Services Architecture”, or OGSA. Each grid service defines five

core interfaces: (1) *service registration*, which allows a grid service to register itself with a service registry; (2) *service location*, which enables service/resource discovery; (3) *service lifecycle management*, defining a core set of grid service stages; (4) *introspection*, which allows grid service capabilities to be dynamically discovered; and (5) *service creation*, allowing new grid services to be dynamically created at runtime.

Although the published anatomy and physiology did much to lay the groundwork for grid software system architectures and have been very widely, often uncritically, cited, they do not readily distinguish grids from traditional software libraries, middleware, and frameworks [6].

2.2. Architectural Recovery

Architectural recovery is the process of elucidating a software system’s architecture, most frequently from source code, but also from other available artifacts. Numerous such techniques have been developed [8]. For the purposes of our study, we are leveraging Focus [9]. Focus clusters system modules recovered via static analysis using a set of rules based on coupling and cohesion.

3. Deconstructing Grid Technologies

We examined the available information from eighteen widely used grid technologies, summarized in Table 1. In the table, the column *PL* indicates the grid technology’s primary implementation language(s); *KSLOC* identifies the source code size of the technology; and *#ofModules* indicates a count of all classes

and any other modules for which member functions could be identified. While we collected a tremendous amount of data in the process, we can only summarize our findings here; the interested reader can find the unabridged results of our study in [10].

To select grid technologies across each of these three types, we applied four criteria: (1) the system should be open source; (2) the system should have available documentation to supplement code analysis; (3) the technology should be actively used; and (4) the system should claim to be a grid.

Since each grid technology we examined was open source, we were able to analyze and visualize its source code using static analysis tools. One challenge we faced was that the studied grid systems were implemented in a number of programming languages. Another challenge was that the static analysis tools vary in quality and tend to give incomplete results (e.g., see [11]). We typically applied multiple tools on the same grid system to ensure that we are correctly

extracting as many static relationships as possible. In the process, we used or attempted to use over 20 static analysis tools (including, e.g., SWAG Kit [12] and SHrIMP [13]). We found that four tools were able to extract the bulk of the static dependencies for each grid technology: Rational Software Architect (RSA) [14], ArgoUML [15], Understand [16], and DoXYGen [17].

We supplemented the information obtained via automated analysis with available documentation and manual inspections of the source code. The above process resulted in architectural models for each studied grid technology.

Our ensuing step involved “shoe-horning” each of the recovered grid components and their interactions (i.e., connectors) into the five-layered grid architecture, using the grid’s anatomy and physiology [1, 2] as a guide. Figure 1 shows the results of this step for *Wings*, *Pegasus*, *Hadoop*, and *iRODS*.

During the shoe-horning process, we repeatedly encountered four types of discrepancies:

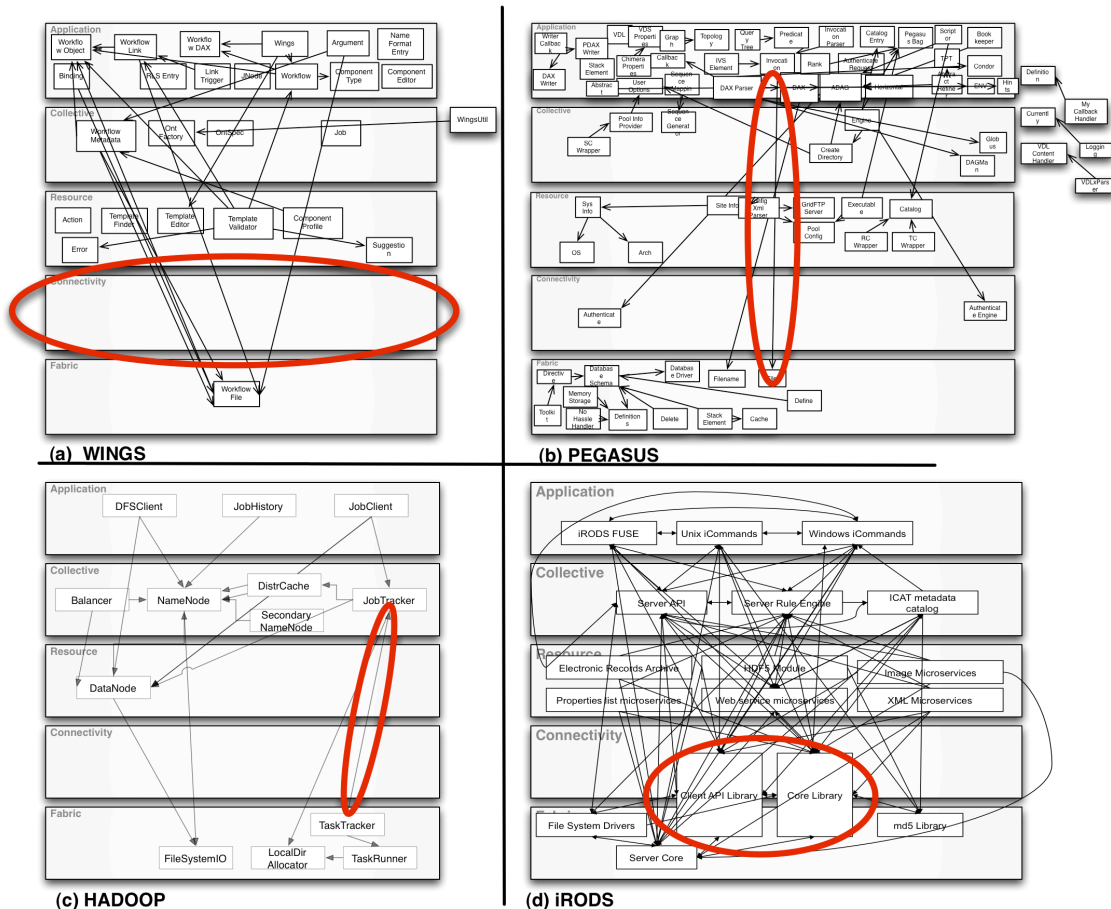


Figure 1. Representative discrepancies identified in four of the studied grid technologies. At this magnification, the reader is not expected to understand the details of these diagrams, but rather to get a general sense of the four architectures.

- (1) *empty layers* – layers identified in the grid’s anatomy that contained no recovered components; an example is highlighted for *Wings* in Figure 1a;
- (2) *skipped layers* – components in one layer make calls to components at least two layers below or above, highlighted for *Pegasus* in Figure 1b;
- (3) *upcalls* – calls made from components in a lower (“servicing”) layer to components in a higher (“client”) layer, such as the example for *Hadoop* highlighted in Figure 1c; and
- (4) *multi-layer components* – components that provided services which, according to the published grid DSSA, belong to two or more layers, e.g., as shown highlighted for *iRODS* in Figure 1d.

In several cases we also identified *orphaned components*, whose exact location in the grid DSSA we were unable to determine. An example is shown for *Pegasus* on the right side of Figure 1b. Orphaned components typically indicated the presence of test classes, templates, and other functionality not intended to be part of the core system.

The studied systems’ sizes varied widely in the number of violations of the grid DSSA. For example, in *Wings* (Figure 1a), which has 97 original modules and around 9 KSLOC, we observed three types of discrepancies: 6 upcalls, 10 skipped layers, and 1 empty layer (*Connectivity*); in addition, *Wings* also had an orphaned component (*WingsUtil*). On the other hand, in *Pegasus*, a much larger system (almost 10x as many SLOC and 7x as many modules), we noted about 70% as many total discrepancies: 6 upcalls and 5 skipped layers; *Pegasus* also had six orphaned components, which we attribute to the fact that, as a larger system, it had more utility modules than *Wings*.

We have found a similar lack of correlation between a grid system’s size and its adherence to the grid DSSA throughout our study. Another example is *Hadoop*, with 66 KSLOC and 1643 modules: it had only 2 upcalls and 6 skipped layer violations. On the other hand, *iRODS*, which was of comparable size (84 KSLOC) but had 10x fewer code-level modules (163), had 36 upcalls, 46 skipped layer violations, and 2 multi-layer components. *iRODS* had the most problematic architecture of the eighteen systems we studied with at least 2x as many upcalls and 4x as many skipped layer violations as the other technologies.

Overall, the most prevalent discrepancies identified were upcalls and skipped layers (a total of 244), as indicated in [10]. Each studied grid technology’s architecture used upcalls (a total of 101 across the eighteen technologies) and all but one (*Condor*) skipped layers (a total of 143 across the eighteen technologies). This suggests that, as conceived in the grid DSSA, the layers share concerns and are ultimately less orthogonal than intended.

Furthermore, as described in [1, 2] and summarized in Section 2, the layered architecture is conceptually abstract, failing to document both the types of grid components that should reside in each layer and the many important interactions between those components. The presence of other identified violations—multi-layer components (a total of 5) and empty layers (a total of 7)—reinforced our conclusion that the existing grid architecture required further refinement.

Acknowledgements

This work is supported by the National Science Foundation under Grant numbers ITR-0312780, CSR-0720612, and SRC-0820170. Effort also supported by the Jet Propulsion Laboratory, managed by the California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

- [1] I. Foster, et al., "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration," Globus Research, Work-in-Progress 2002.
- [2] C. Kesselman, et al., "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *Intl' Journal of Supercomputing Applications*, pp. 1-25, 2001.
- [3] C. Mattmann, et al., "A Software Architecture-Based Framework for Highly Distributed and Data Intensive Scientific Applications," In Proc. *ICSE*, 2006.
- [4] C. Mattmann, et al., "GLIDE: A Grid-based, Lightweight, Infrastructure for Data-intensive Environments," In Proc. *EGC*, 2005.
- [5] W. Tracz, et al., "Software Development using Domain-Specific Software Architectures," *ACM SEN*, pp. 27-38, 1995.
- [6] C. Mattmann, et al., "Unlocking the Grid," In Proc. *CBSE*, 2005.
- [7] R. N. Taylor, et al., *Software Architecture: Foundations, Theory and Practice*: John Wiley & Sons, 2009.
- [8] R. Koschke, "Rekonstruktion von Software-Architekturen," *Informatik-Forschung und Entwicklung*, vol. 19, pp. 127-140, 2005.
- [9] N. Medvidovic and V. Jakobac, "Using Software Evolution to Focus Architectural Recovery," *JASE*, vol. 13, pp. 225-256, 2006.
- [10] "Grid Systems Software Architecture - Supplementary Website, <http://sunset.usc.edu/~softarch/grids/>," 2008.
- [11] S. E. Sim, et al., "On using a benchmark to evaluate C++ extractors," In Proc. *IWPC*, 2005.
- [12] "SWAG Kit, <http://www.swag.uwaterloo.ca/swagkit/index.html>," 2008.
- [13] M.-A. Storey, "ShriMP views: an interactive environment for exploring multiple hierarchical views of a Java program," in *ICSE 2001 Workshop on Software Visualization*, 2001.
- [14] "IBM - Rational Software Architect, <http://www-01.ibm.com/software/awdtools/architect/swarchitect/>," 2008.
- [15] J. Robbins and D. Redmiles, "Cognitive support, UML adherence, and XMI interchange in Argo/UML," *Information and Software Technology*, vol. 42, pp. 79-89, 2000.
- [16] "Understand - Source Code Analysis and Metrics, <http://www.scitools.com/products/understand/>," 2008.
- [17] "Doxygen, <http://www.stack.nl/~dimitri/doxygen/>," 2008."