Unlocking the Grid

Chris A. Mattmann[1,2], Nenad Medvidovic[2], Paul M. Ramirez[1,2] and Vladimir Jakobac[2]

[1]Jet Propulsion Laboratory, 4800 Oak Grove Drive, M/S 171-264
Pasadena, CA 91109, USA
{chris.mattmann,paul.ramirez}@jpl.nasa.gov
[2]University of Southern California
Los Angeles, CA 90089, USA,
{mattmann,neno,pmramire,jakobac}@usc.edu

Abstract. The grid has emerged as a novel paradigm that supports seamless cooperation of distributed, heterogeneous computing resources in addressing highly complex computing and data management tasks. A number of software technologies have emerged to enable "grid computing". However, their exact nature, underlying principles, requirements, and architecture are still not fully understood and remain under-specified. In this paper, we present the results of a study whose goal was to try to identify the key underlying requirements and shared architectural traits of grid technologies. We then used these requirements and architecture in assessing five existing, representative grid technologies. Our studies show a fair amount of deviation by the individual technologies from the widely cited baseline grid architecture. Our studies also suggest a core set of critical requirements that must be satisfied by grid technologies, and highlight a key distinction between "computational" and "data" grids in terms of the identified requirements.

1 Introduction

The *grid* is an emerging paradigm concerned with enabling heterogeneous organizational entities to share computing resources (both hardware and software), data, security infrastructure, and the like [4]. Additionally, the grid's goal is to allow such organizations to operate in a coordinated fashion to solve very complex scientific and information management problems [4,14]. Because of this, the grid has become an area of significant interest to computing researchers and practitioners, and a number of open source and standards-based grid infrastructure implementations exist and have commercial backing.[1]

Recently, in collaboration with NASA's Jet Propulsion Laboratory (JPL) our research group decided to port JPL's OODT grid technology [6] onto our Prism-MW middleware platform for mobile and resource constrained devices [25]. The goal was to significantly reduce OODT's footprint and "bring the grid into one's pocket". The ex-

1. For exposition purposes, we will use the phrases "grid infrastructure", "grid technology", "grid solution", "grid platform", and "grid system" interchangeably in this paper

G.T. Heineman et al. (Eds.): CBSE 2005, LNCS 3489, pp. 322-336, 2005.
© Springer-Verlag Berlin Heidelberg 2005

ercise was successful, JPL deemed our prototype, GLIDE, quite promising, and we decided to document the experience in a research paper [5], which we submitted to a workshop on middleware technologies.

When we received the reviews for the paper, one particular comment caught our attention. One reviewer was unimpressed with what we had done in part because, in the reviewer's words, OODT "is in itself a very simple class framework"; another reviewer also alluded to this! This was very surprising, given that OODT has been a highly successful grid technology, deployed both within NASA and externally with the National Cancer Institute, and was the runner-up for NASA's Software System of the Year award in 2003.

The grid literature is very rich in general ("reference") requirements a grid platform should satisfy [3, 14, 13], and also details its target ("reference") architecture [15, 21]. Based on this, we had gone with the assumption that a grid technology can be relatively easily distinguished from "something else". However, a review of the OODT documentation revealed that no such distinguishing features were obviously stated. We then studied the documentation accompanying several other grid solutions and found that the same holds for them. Thus, the comment we received, from experts in the area, raised three questions that directly motivated the study on which we will report in this paper:

1. What, in fact, makes a software system a grid technology?
2. What, if any, is the difference between a grid technology, a middleware platform, a software library, and a class framework?
3. Are existing systems that claim to enable grid computing *bona fide* grid technologies?

In order to answer these questions, we decided to recover, study, and compare the architectures of a number of existing grid technologies. Specifically, we chose five such technologies, including Globus, the most widely used grid system, as well as OODT and GLIDE, the two systems that prompted our study in the first place. In principle, the only requirement in selecting the candidate grid technologies was that they be open source. Since OODT was compared to a class framework, we also decided to restrict our study to object-oriented grids. While we decided to apply a particular software architecture recovery technique in our study [23], the technique is representative of a number of architectural recovery approaches, and we do not believe that it significantly influenced our results. The recovered architectures were "interpreted" with the help of the reference requirements and reference architecture for grid systems we gathered from existing literature.

In this paper, we present the details of this study, and the lessons we learned in the process. The overall conclusion of our work has been that grid technologies, including OODT and GLIDE, do in fact adhere to a specific architecture and are thus quite different from software libraries and class frameworks. At the same time, our study also revealed that several aspects of the published grid reference requirements and architecture are overly general and open ended, so much so that it was at times difficult to imagine what a given grid solution would have to do to deviate from them. Based on this, we suggest some improvements to the current state of the practice in grid computing infrastructures.

The remainder of the paper is organized as follows. Section 2 outlines our background research which resulted in reference requirements and a reference architecture for grid systems, and discusses related work in the area of architectural recovery. Section 3 describes the approach we have taken in our study, while Section 4 summarizes the results of applying the approach on five off-the-shelf (OTS) grid technologies. Section 5 highlights the lessons we have learned in the process and suggests possible future work in the area of grids. We then conclude the paper in Section 6.

2 Background and Related Work

In order to effectively study grid technologies, we needed to identify their overarching requirements and shared architectural traits. The existing grid literature contains four separate studies that attempt to provide such information [3, 4, 14, 18]. However, in addition to being dispersed, this information was presented in widely differing ways, at times ambiguous, influenced or obscured by details of particular grid solutions, and even contradictory. Our task thus consisted of locating, compiling, rephrasing (if necessary), and consolidating the requirements.

Particularly helpful in this task was the seminal study of grids by Kesselman et al. [4]. This study provides a rich target set of requirements by exploring a suggested five-layer grid reference architecture. Each layer in the architecture defines services (i.e., software components) that should satisfy particular requirements (including QoS, characteristics, and capabilities) mentioned in the description of the layer. However, many of these requirements are not explicitly called out and had to be "distilled" from the text. In addition, some requirements overlap, while others span architectural layers.

A particular class of grid solutions, called data grids, provides services primarily targeted at managing data and metadata resources. Chervenak et al. [3] identify four guiding principles for data grids: *mechanism neutrality*, *policy neutrality*, *compatibility with grid infrastructure*, and *uniformity of information infrastructure*. However, we found the natural language presentation of these principles ambiguous, especially when we initially tried to assess the conformance of grid solutions to them. Moreover, there is no mapping of the principles to constituent architectural components in grid solutions. We thus had to rephrase and interpret them. Our further research also identified additional requirements for data grids involving replica management, metadata management, and interfaces to heterogeneous storage systems [2, 10].

A significant aspect of our work is the recovery of grid platforms' architectures from their implementations. A number of architecture recovery approaches have been developed in the past decade (e.g. [7, 9, 11, 12, 29]). They typically analyze dependencies among a system's implementation modules (e.g., procedures or classes) to cluster them into higher-level components. A more detailed overview of these approaches can be found in [23].

Recently, a series of studies has been undertaken by Holt et al. to recover the architectures of several open-source applications [15, 19]. Similarly to our approach, the approach taken in these studies has been to come up with an "as-intended" (i.e., reference) architecture by consulting a system's designers and its documentation, and use it as the

basis for understanding the system's "as-implemented" architecture recovered from the source code.

Table 1: Reference Requirements for Grids

	Requirement	Impacted Layer
1	Share resources across dynamic and geographically dispered organizations	Collective
2	Enable single sign-on	Connectivity
3	Delegate and authorize	Connectivity
4	Ensure access control	Connectivity
5	Ensure application of local and global policies	Fabric
6	Control shared resources	Collective
7	Coordinate shared resources	Collective
8	Ensure "exactly once" level of reliability service	Connectivity, Application Resource
9	Use standard, "open" protocols and interfaces	Collective, Resource, Connectivity
10	Provide ability to achieve non-trivial QoS	Application, Resource, Collective
11	Ensure neutrality of data sharing mechanism	All layer's implementation
12	Ensure neutrality of data sharing policy	Collective, Resource, Connectivity, Fabric
13	Ensure compatibility with Grid infrastructure	Possibly all layers
14	Provide uniform information infrastructure	Application, Resource, Collective
15	Support metadata management	Resource
16	Interface with heterogeneous storage systems	Fabric
17	Provide the management of data replicas	Fabric, Resource, Collective

Another related architecture recovery approach is Dynamo-1 [20], which focuses on middleware-based software applications. It combines the use-case modeling aspect of Focus [23], the approach we have adopted in this work, with the filtering and clustering approach of PBS [26]. Dynamo-1 differs from our approach in that its goal is only to recover the architectures of the *applications* hosted on top of the given middleware infrastructure, while our goal is to analyze and recover the architecture of the grid infrastructure itself.

3 Approach

The approach that we used in our case studies is depicted in Figure 1. It involves three high-level activities. The first activity has two sub tasks (1a and 1b) that were conducted only once, and independently of the other activities. The remaining activities (and their subtasks) were conducted iteratively in each grid technology we studied. We detail our approach below.

3.1 Reference Architecture and Requirements

After studying the available grid literature as outlined in Section 2, we identified the *de facto* reference architecture for grid systems [4] (Step 1a in Figure 1). The architecture consists of five layers, each of which relies on the services of its subordinate layer(s).

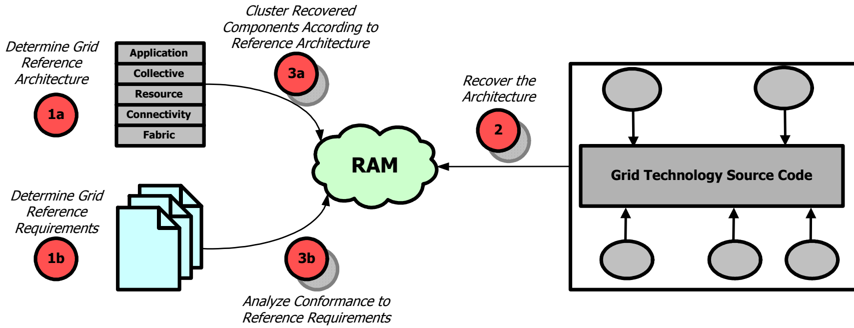

Figure 1 Our approach to studying grid technologies.

The *application layer* represents the software system built using the grid technology, and it relies on the functionality for coordinating system resources available via the *collective layer*. The collective layer coordinates and manages resources exposed at the *resource layer*. The top three layers directly rely on the *connectivity layer* for secure and distributed communication. The bottom layer is the *fabric layer*, providing standard interfaces to heterogeneous system resources, such as file systems and device drivers.

In addition to the reference architecture, we also identified a representative set of high-level requirements for grids (Step 1b in Figure 1) in the manner outlined in Section 2. Due to space restrictions, these requirements are only briefly summarized in Table 1. We have also identified the reference architecture layer(s) that are likely impacted by each requirement.

Both the reference architecture and reference requirements formulate the targets for our study and are used to inform our architecture recovery efforts which we describe below.

3.2 Architecture Recovery

Architectural recovery (Step 2 in Figure 1) involves automated examination of the source code to extract an "as-implemented" architectural model for each grid technology. We refer to this model as the *recovered architectural model*, or RAM. We used the Focus architectural recovery approach [23] in this step. Focus was selected because it is specifically geared to object-oriented systems, while it has been shown to produce comparable results to other recovery approaches. The key steps of Focus are briefly summarized below. Additional details may be found in [23].

Focus relies on an OTS source code extraction tool (we used Rational Rose in our study) to generate class diagrams from the given grid technology's code. Once a class diagram has been extracted from the code, a set of automatable clustering rules are applied iteratively to group individual classes into higher-level components. These rules include grouping classes that share aggregation, generalization, and two-way relationships, grouping clusters of classes that are isolated from the rest of the system, identifying "important" classes that have many incoming and outgoing links, and so forth. Focus identifies two kinds of components: processing and data. It also attempts to identify the key communication elements in a system.

Once these rules cannot be applied on the system's (clustered) class diagram any longer, Focus has produced the RAM. It is important to acknowledge that this architectural model may have several limiting properties:

- *The RAM may not be complete* - Any clustering-based approach may fail to identify all system components. The RAM may also ignore component interaction characteristics (i.e., connectors). Further, the RAM may not provide any insight into the system's legal configurations.
- *The RAM may not conform to the system's intended architecture* - Over time, the system's implementation may have (significantly) deviated from the designers' original intentions. This is referred to as architectural drift or erosion [27].
- *There is no obvious relationship between the RAM and the system's requirements* - The only input to Focus (and many other architectural recovery approaches) is the source code. As such, the recovered architecture does not identify the requirements each component is intended to fulfill.

3.3 RAM Reconciliation

At this point, we have the reference requirements and architecture for grid technologies, as well as the RAM for the particular grid system. Since the RAM may deviate from the reference architecture, the next step in our approach is to reconcile the RAM and the reference architecture, i.e., to place the components identified in the RAM into specific layers of the architecture (Step 3a in Figure 1).

For each RAM component, we try to identify its counterpart in the reference architecture. To do so, we examine (1) any information about the component's functionality from the documentation of the grid system under study, (2) the component's relationships with other components, and possibly (3) the description of similar components in the grid literature.

Once the decision is made to place a component in a given architectural layer, the relationships among the components are examined more closely to identify two types of inconsistencies: (1) the grid reference architecture [4] implies that, with one exception, the layers are opaque, such that components in a given layer can only access services of the layer immediately below; and (2) the layered architectural style prohibits components from making "up-calls". At this point we also note any additional discrepancies, such as our inability to assign a component to any layers, "invalid" or unexpected dependencies among components, and so on.

Finally, since different grid technologies may have different foci (e.g., computational vs. data grids, or high-performance computing vs. pervasive grids) and may approach the problem differently, another relevant piece of information is the degree to which the reconciled RAM adheres to the reference requirements (Step 3b in Figure 1). The goal of this activity is to identify the requirement(s) satisfied by each component, including the components we were unable to fit into the reconciled RAM. We again try to identify any discrepancies in the placement of components in the architectural layers based on the location guidelines shown in the right-hand column of Table 1.

4 Case Studies

We have used the approach detailed above to study five different grid technologies, selected based on the following criteria. First, the technology should be open-source because we needed the ability to perform architectural recovery from the source code. Second, the technology should be object-oriented because, as discussed in the Introduction, one of our objectives was to discover the relationship of grid solutions and class frameworks. Third, we required at least some level of documentation to aid us in determining the functionality of the recovered grid components. We do not feel this requirement to be particularly limiting since the documentation could be as simple as an HTML page (as was indeed the case with the JCGrid study discussed below). Fourth, we wanted to study grid technologies that are used in "legitimate" industrial and/or academic projects in order to ensure the relevance of our results. Finally, we wanted the set of studied systems to include OODT and GLIDE because, as discussed in the Introduction, they were the direct motivators for this study. It should be noted that OODT does satisfy all of our criteria; however, GLIDE is currently being evaluated and thus can be argued not to satisfy the fourth criterion at this time.

In this section, we detail our studies of OODT and Globus, and summarize the results of the remaining three studies. Additional details on all five studies can be found in [22].

4.1 OODT

OODT [6] is a grid infrastructure developed at JPL in support of scientific, data-intensive grid systems. OODT's implementation consists of approximately 14,000 SLOC. The initially recovered OODT class diagram, shown in Figure 2, contained 320 classes. For the most part, it was a densely connected graph, but it also contained approximately 40 classes with no recognized relationships to other classes in the system (shown isolated in the bottom-left portion of Figure 2). UML generalization and interface relationships were most prevalent, with many classes implementing at most one interface.

We applied the iterative clustering rules of Focus on the class diagram to arrive at the OODT RAM. The RAM comprised 38 processing and data components, along with the identified relationships (i.e., connectors) between each component pair. With the help of the component descriptions in OODT's conceptual architecture [6], we were able to place 24 of the 38 RAM components into the layered grid reference architecture with relative ease. The remaining components had to be "shoehorned" by examining ad-

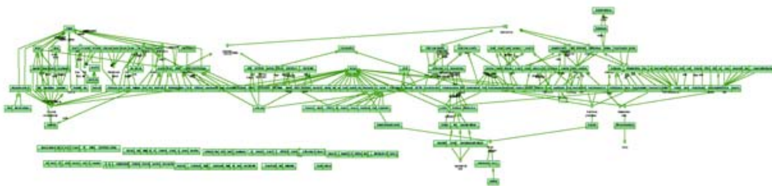

Figure 2 Initial class diagram of OODT. Due to its size and complexity, at this magnification the diagram is shown only for illustration

ditional OODT documentation [21] and, in a few cases, the OODT source code. The final result of this process is shown in Figure 3.

Of note is the fact that only one component was placed in the connectivity layer of the reference architecture. The reason is that OODT leverages third-party middleware platforms (CORBA and RMI) to support distribution, and those are considered external to its code base. There are also several deviations from the reference architecture:

1. Several components in the fabric layer (*ProductServicePOA*, *ProductServiceAdaptor*, and *CORBA_Archive_ServicePOA*) communicate with the *ExecServer* component in the resource layer. The fabric layer components not only cross two layer boundaries, but also make up-calls to perform this communication.
2. Components in the application layer (*ConfiguraitonBean* and *SearchBean*) communicate with the *Configuration* component in the connectivity layer, crossing three layer boundaries. Similarly, the *ProfileClient* and *ProductClient* components in the application layer traverse two and four layer boundaries, respectively, to communicate with their server components.
3. The *Utilities* component (shown in the upper-right of Figure 3) was identified by Focus as comprising classes with no recognizable relationships with other classes. We were unable to determine its correct placement in the reference architecture.

Our analysis of the architecture shown in Figure 3 suggests that OODT satisfies most of the reference requirements specific to data grids[1]. The lone exception is its lack of

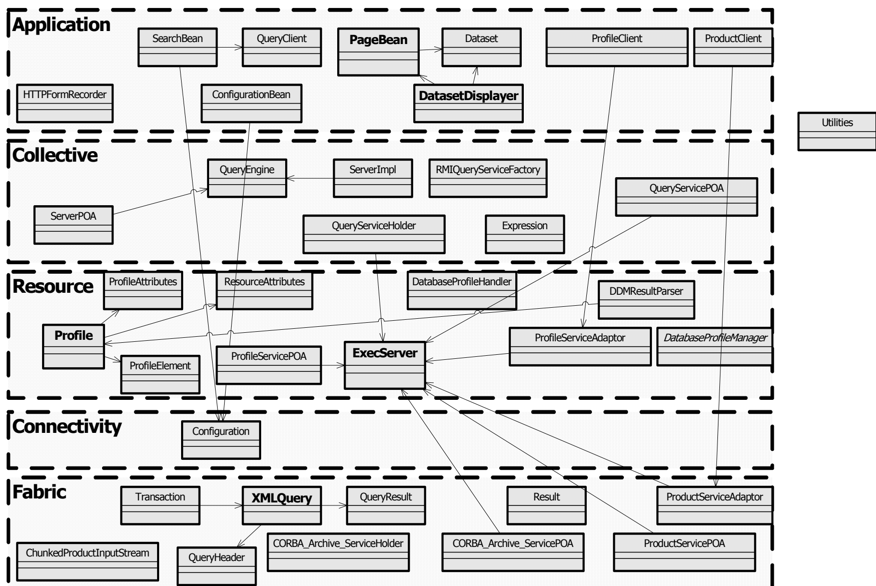

Figure 3 Mapping OODT's RAM onto the grid reference architecture

1. Recall the discussion in Section 2 and corresponding requirements 11-17 in Table 1.

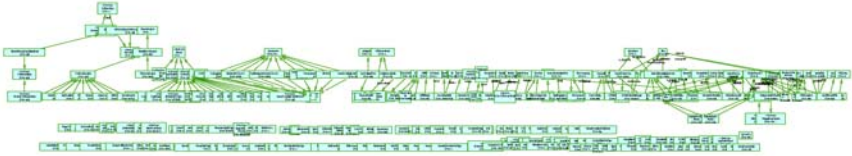

Figure 4 Initial class diagram of Globus.

support for replica management (requirement 17 in Table 1). In addition, OODT fails to address two other reference requirements: single sign-on (requirement 2) and "exactly once" level of reliability service (requirement 8). It appears that this stems from OODT's reliance on CORBA and RMI for such basic services.

4.2 Globus

The Globus toolkit [14] has been used successfully across a number of projects [2, 4, 8, 10, 16], and can be considered to be the *de facto* standard for grid implementations. The initially recovered class diagram of Globus is shown in Figure 4 for illustration. Globus consists of 864 classes and approximately 55,000 SLOC; it was the largest and most complex grid technology that we studied. Similarly to OODT, Globus also has a set of classes (around 60) that share no recognizable relationship with any other classes (appearing at the bottom of Figure 4).

After applying Focus on Globus' source code, we arrived at the Globus RAM. The RAM contained 86 components, 50 of which were identified as data components. Most processing components contained at least one relationship, typically a UML association, with another class.

Relying on the documentation that was included with the Globus core distribution [28], along with our study of the existing Globus literature [2, 4, 10, 13, 14], we were able to place 81 of the 86 Globus RAM components into the layered grid reference architecture. This high percentage was unsurprising since Globus is the realization, and served as the direct inspiration, of the reference architecture presented by Kesselman et al. in [4]. Still, the architecture we recovered did deviate from the proposed reference architecture as discussed below.

As shown in Figure 5 most of the components found their way into the resource and connectivity layers, while only two components were assigned to the fabric layer. Similarly to OODT, Globus also relies on a third-party distributed communication solution: Apache's AXIS implementation of the SOAP protocol. The Globus release includes AXIS; hence the large number of components in the connectivity layer. Globus' major deviations from the reference architecture are as follows:

1. The *Logging* component appears to have a home in both the collective and resource layers of the reference architecture. Similarly, the *Map* data component that we placed in the collective layer actually permeates several of the other layers, and may in fact belong somewhere else. Given that it is a basic data compo-

nent (indeed, implemented as a *Hashtable*), we could not confidently discern its requisite architectural layer.

2. The *JavaClassWriter* component in the fabric layer appears to be making an up-call two layers above to the *ServiceEntry* component in the resource layer. Similarly, the *Java2WSDL* component in the resource layer is making a two-layer up-call to the *CLOptionDescripton* component in the application layer.
3. There were five components, including a *Utilities* processing component and an *Exception* data component, which we were unable to assign to any layers of the reference architecture. This was because the given component appeared to belong to more than one layer, we could not find sufficient documentation for it, and/or it did not have enough relationships in the RAM to positively classify it to a particular layer.

Globus addresses the first 10 grid reference requirements identified in Table 1; however, it does not satisfy all of the remaining requirements, which are specific to data grids. In particular, we discovered that Globus does not natively support requirements 14-16. These capabilities are provided by components built on top of the Globus grid infrastructure, such as the Metadata Catalog Service [10], and the Replica Location Service [2] components.

4.3 Summary of Remaining Studies

Due to space limitations, we only summarize the remaining three case studies here; their complete treatment is provided in [22].

GLIDE [5] is the grid technology that directly motivated our work presented in this paper. It is a lightweight grid infrastructure for data-intensive environments. GLIDE's goal is to extend the grid paradigm to the emerging decentralized, resource-constrained, embedded, autonomic and mobile environments.

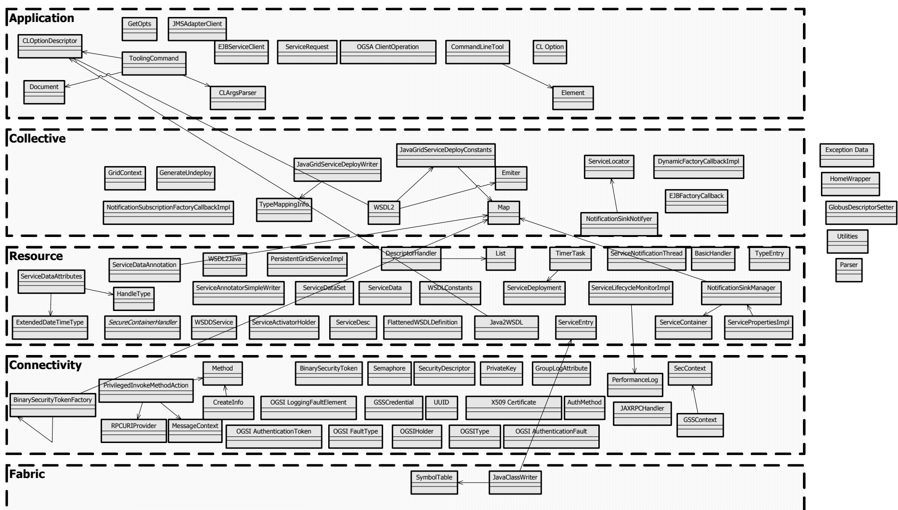

Figure 5 Mapping Globus's RAM onto the grid reference architecture

Dspace [1] is an open source grid system, jointly developed by MIT Libraries and Hewlett-Packard. It is a distributed digital repository system that captures, stores, indexes, preserves, and redistributes the research material of a university in digital formats.

Finally, JCGrid is a computational grid platform developed in Java and available via open source on SourceForge [30]. JCGrid allows one to split CPU-intensive tasks among multiple workstations. It has been used in several significant applications to date.

The implementation sizes of the three technologies ranged from 2,100 SLOC distributed over 61 classes in GLIDE, to 24,000 SLOC and 217 classes in DSpace. All three technologies violated the reference architecture. Some discrepancies include communication spanning the application and connectivity layers in GLIDE, communication across all five layer boundaries in JCGrid (the most significant such deviation observed in the five case studies), and a component (*WorkflowManager*) spanning both the collective and resource layers in DSpace.

The task of mapping the components to the reference architecture was relatively straight forward in the cases of GLIDE and JCGrid: GLIDE's components were similar to OODT's, while JCGrid RAM's 37 components nicely conformed to different architectural layers. DSpace was much more challenging in this regard: many of its components appeared to span several architectural layers and we could not confidently ascertain their appropriate "homes".

In terms of the reference requirements, one observation was that neither GLIDE nor DSpace supports replica management (requirement 17 from Table 1), even though they claim to be data grid solutions. Finally, JCGrid's reliance on a *GridServer* component to manage policy and access control does not bode well for its ability to support application of local and global policies (requirements 3-5).

5 Discussion

Our objective in conducting this study was to clearly identify what distinguishes a *bona fide* grid technology from other utility software, such as "ordinary" middleware platforms, software libraries, and frameworks. We believe that we have achieved a qualified success in this endeavor, but that we have also identified several deficiencies in the current level of understanding of the grid. These deficiencies can, in turn, form a coherent research agenda for the grid community.

Grid technologies can be thought of most appropriately as specialized middleware platforms that share a reasonably well defined reference architecture. In other words, grid technologies are an example of domain-specific software architectures [17], for the domain of grid computing.

As such, grid systems have little in common with software libraries, although they may indeed provide useful services in the form of libraries. Similarly, while the implementation of each of the five technologies we studied may be looked at as a framework of object-oriented classes that is specialized and instantiated to solve a particular prob-

lem, we believe that to be the wrong abstraction in this case. The fact that a given technology is designed and implemented in, say, Java is incidental; a number of successful grid technologies have in fact been designed and implemented using the procedural paradigm and languages (e.g., C). The key property of a grid technology is its satisfaction of a well defined set of requirements via functional services that are distributed across five well defined architectural layers.

Of course, as can be gleaned from Section 4, the preceding statement is only partially true. The existing grid technologies vary widely in the selection of requirements they choose to satisfy, as well as in the functional services they provide in the form of components. If we consider the five grid technologies we studied in depth, they covered a very broad range in terms of source code size and implementation class complexity. For example, GLIDE was implemented in slightly over 2,000 SLOC, while the current implementation of Globus is at about 55,000 SLOC. Likewise, GLIDE's entire implementation comprises 61 classes, while Globus has over 14 times as many.

This discrepancy can be partly attributed to the differences in the design choices and foci of the different grid technologies. For example, Globus subsumes a relatively large third-party middleware platform (AXIS) and provides numerous utilities to its users; on the other hand, GLIDE leverages a much smaller middleware platform and provides only basic grid services. Furthermore, each grid technology we studied differs in the adopted distributed communication mechanism, and the type and degree of support provided to application developers, as summarized in the below table.

	Communication Mechanism	**Application Development Support**
OODT	Remote method invocation	Object-Oriented
Globus	SOAP Publish and Subscribe	Web-services based
GLIDE	Event-based and publish-subscribe	Software architectural style-based
DSpace	Client-server over HTTP	Object-Oriented
JCGrid	Client-server with asynchronous invocation	Object-Oriented

In addition to the above, at least to some extent the discrepancies found across the grid technologies are a by-product of the reference requirements each development group has chosen to satisfy. There are currently no guidelines for which requirements are mandatory and which are optional. Based on our study, it appears that requirements numbered 1, 5-7, and 9-10 in Table 1 are mandatory (i.e., every grid technology must satisfy those), while the rest are optional. Moreover, if we consider more carefully the intended uses of the grid systems, we can identify a finer distinction, one that pinpoints the difference between "computational grid" and "data grid" systems. As indicated in Section 2, this distinction has been widely used in literature (e.g., [3, 4, 10, 18]), but has not been carefully explained or justified to date. DSpace's, OODT's, and GLIDE's primary stated objective is, in fact, to support data grids. Then, based on the results of our studies we hypothesize that requirements numbered 11-12 and 14-17 must be satisfied

in addition to the above set of mandatory grid requirements in order to support data grids.

If we shift our focus more toward the architectures of the studied grid systems, again some interesting observations emerge. Even though these systems vary widely in their size, complexity, and specific focus, for the most part their components map rather nicely to the five-layer reference architecture. Upon closer inspection, we believe this to be true for several reasons that must be addressed by the grid community.

First, *the requirements for grid systems are very broad*, and generally applicable to a number of middleware solutions. Several of the grid requirements involve basic middleware QoS requirements such as security, dependability, marshalling of data, and the use of standard, open interfaces. These requirements give little help in distinguishing a grid solution from "something else"; alternatively, given such generally applicable requirements, it is difficult *not* to provide at least some grid capabilities.

Secondly, *there is overlap between grid layers*. An example is the difference between the resource and collective layers, where one layer coordinates individual resources and the other layer multiple resources. In practice, it has been difficult to determine the layer to which a given component belongs. For example, if only a single resource of a given kind exists, in which layer should the corresponding component be placed, or do there still need to be two separate components, one in each layer?

Third, *grid technologies regularly violate the reference architecture*. Specifically, nearly all of the grid systems that we studied fail to conform to the restrictions of the layered architecture style. Violations include component communication spanning multiple layers, up-calls, and dependencies between layers that were not specified in the reference architecture. This is at least in part caused by the haphazard way in which the requirements and architectures of existing grid systems are captured. We believe that appropriate use of architectural formalisms, such as architecture description languages [24], would provide the needed descriptive power as well as rigor to support the validation of each of these systems against the constraints specified in their reference architecture.

Finally, it is also evident that, due to the broad definition of what constitutes a grid technology, *interoperability between grid solutions poses a key challenge*. Even conformance to the recently adopted Open Grid Services Architecture (OGSA) [14] does not guarantee interoperability between grid middleware systems. This is in part evidenced by OGSA's lack of backward compatibility with previous Globus systems, which directly influenced OGSA. Many questions still remain, such as, what data do the grid services exchange and how is it described? OGSA and the recently announced Web Service Resource Framework (WS-RF) represent initial steps towards remedying this problem, but the problem remains wide open.

6 Conclusion

Our study of grid technologies has corroborated some of the claims made in grid literature, while suggesting refinements to others. In particular, we found the reference grid

architecture [4] a useful baseline for comparing disparate grid solutions, especially since their recovered architectures (i.e., RAMs) were quite divergent. Furthermore, the reference requirements we distilled from literature certainly helped to improve our understanding of the grid. Together, the architecture and requirements suggest a tangible distinction between grid technologies on the one hand, and commonly used software notions such as middleware, libraries, and frameworks on the other. Another distinction rendered more concrete by our study is between computational and data grids.

At the same time, one conclusion of our study is clear: the answer to the question "what makes a grid system a grid system?" has many possible answers. This is not necessarily a drawback, as it allows developers of a given grid platform to tailor its functionality, and to some extent its architecture, to the needs at hand. At the same time, we argue that this open-endedness may become an impediment to the on-going standardization efforts and the claimed goal of grid platform interoperability.

7 Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant Numbers CCR-9985441 and ITR-0312780. Effort also supported by the Jet Propulsion Laboratory, managed by the California Institute of Technology.

8 References

[1] DSpace at MIT. DSpace: An Open Source Dynamic Digital Repository. *D-Lib Magazine*, 9(1), January 2003.

[2] A. Chervenak et al. Giggle: A Framework for Constructing Scalable Replica Location Services. In *Proc. of IEEE Supercomputing Conference*, pp. 1-17, 2002.

[3] A. L. Chervenak et al. The Data Grid: Towards an Architecture for the Distributed management and analysis of Large Scientifc Datasets. *Journal of Network and Computer Applications*, pp. 1-12, 2000.

[4] C. Kesselman et al. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputing Applications*, pages 1-25, 2001.

[5] C. Mattmann et al. GLIDE: A Grid-based, Lightweight Infrastructure for Data-intensive Environments. *Technical Report* USC-CSE-2004-509, University of Southern California, June 2004.

[6] D. Crichton et al. A Science Data System Architecture for Information Retrieval, in *Clustering and Information Retrieval*, W. Wu, H. Xiong, S. Shekhar (Eds.):pages 261-298. Kluwer Academic Pubishers, December 2003.

[7] D. R. Harris et al. Extracting Architecture Features from Source Code. *Automated Software Engineering*, vol. 3, no. 1/2, pp.109-138, 1996.

[8] E. Deelman et al. Grid-Based Galaxy Morphology Analysis for the National Virtual Observatory. In *Proc. of IEEE Supercomputing Conference*, p. 47, 2003.

[9] G. Murphy et al. Software Refection Models: Bridging the Gap between Design and Implementation. *IEEE Transactions on Software Engineering*, 27(4):364-380, 2001.

[10] G. Singh et al. A Metadata Catalog Service for Data-intensive applications. In *Proc. of IEEE Supercomputing Conference*, pg. 33, 2003.

[11] G.Y. Guo et al. A Software Architecture Reconstruction Method. In *Proc. of First Working IFIP Conference on Software Architecture*, pp. 15-34, 1999.

[12] H. Gall et al. Object-Oriented Re-Architecting. In *Proc. of 5th European Software Engineering Conference*, pp. 499-519, 1995.

[13] I. Foster et al. Grid services for Distributed Systems Integration. *IEEE Computer*, pp. 37-46, June 2002.

[14] I. Foster et al. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. *Work in progress*, Globus Research, 2002.

[15] I. T. Bowman et al. Linux as a Case Study: Its Extracted Software Architecture. In *Proc. of International Conference on Software Engineering*, pp. 555-563, 1999.

[16] J. Blythe et al. Transparent Grid Computing: A Knowledge-Based Approach. In *Proc. of Innovative Applications of Artificial Intelligence (IAAI)*, pp. 57-64, 2003.

[17] W. Tracz et al. Software development using domain-specific software architectures. *ACM Software Engineering Notes*, pages 27-38, 1995.

[18] I. Foster. What is the Grid?: A three point checklist. *GridToday*, 1(6), 2002.

[19] A.E. Hassan and R.C. Holt. A Reference Architecture for Web Servers. In *Proc. of Working Conference on Reverse Engineering*, pg. 150, 2000.

[20] I. Ivkovic and M.W. Godfrey. Architecture Recovery of Dynamically Linked Applications: A Case Study. In *Proc. of IEEE International Workshop on Program Comprehension*, pp. 178-186, 2002.

[21] S. Kelly. OODT Web Documentation. web site: http://oodt.jpl.nasa.gov, 2004.

[22] C. Mattmann. Recovering the Architectures of Grid-based Software Systems. web site: http://www-scf.usc.edu/~mattmann/GridMiddlewares/, 2004.

[23] N. Medvidovic and V. Jakobac. Using Software Evolution to Focus Architectural Recovery. *Automated Software Engineering*, to appear.

[24] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70-93, 2000.

[25] M. Mikic-Rakic and N. Medvidovic. Adaptable Architectural Middleware for Programming-in-the-Small-and-Many. In *Proc. of ACM/IFIP/USENIX International Middleware Conference*, pp. 162-181, 2003.

[26] Univ. of Waterloo. PBS: Portable Bookshelf. web site: http://swag.uwaterloo.ca/pbs/intro.html, 2004.

[27] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architecture. *ACM Software Engineering Notes*, 17:4, 1992.

[28] T. Sandholm and J. Gawor. Globus Toolkit 3 Core - A Grid Service Container Framework. *Technical report*, Argonne National Laboratory, 2003.

[29] K. Sartipi and K. Kontogiannis. Pattern-based Software Architecture Recovery. In *Proc. of 2nd ASERC Workshop on Software Architecture*, 7 pages, 2003.

[30] Sourceforget.net: Project Info - Java Grid Computing. web site: http://sourceforge.net/projects/jcgrid, 2004.