A Software Architecture-Based Framework for Highly Distributed and Data Intensive Scientific Applications

Chris A. Mattmann[1, 2] Daniel J. Crichton[1] Nenad Medvidovic[2] Steve Hughes[1]

[1]Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109, USA
{dan.crichton,mattmann,steve.hughes}@jpl.nasa.gov

[2]Computer Science Department
University of Southern California
Los Angeles, CA 90089, USA
{mattmann,neno}@usc.edu

ABSTRACT
Modern scientific research is increasingly conducted by virtual communities of scientists distributed around the world. The data volumes created by these communities are extremely large, and growing rapidly. The management of the resulting highly distributed, virtual data systems is a complex task, characterized by a number of formidable technical challenges, many of which are of a software engineering nature. In this paper we describe our experience over the past seven years in constructing and deploying OODT, a software framework that supports large, distributed, virtual scientific communities. We outline the key software engineering challenges that we faced, and addressed, along the way. We argue that a major contributor to the success of OODT was its explicit focus on software architecture. We describe several large-scale, real-world deployments of OODT, and the manner in which OODT helped us to address the domain-specific challenges induced by each deployment.

Categories and Subject Descriptors
D.2 Software Engineering, D.2.11 Domain Specific Architectures

Keywords
OODT, Data Management, Software Architecture.

1. INTRODUCTION
Software systems of today are very large, highly complex, often widely distributed, increasingly decentralized, dynamic, and mobile. There are many causes behind this, spanning virtually all facets of human endeavor: desired advances in education, entertainment, medicine, military technology, telecommunications, transportation, and so on.

One major driver of software's growing complexity is scientific research and exploration. Today's scientists are solving problems of until recently unimaginable complexity with the help of software. They also actively and regularly collaborate with colleagues around the world, something that has become possible only relatively recently, again ultimately thanks to software. They are collecting, producing, sharing, and disseminating large amounts of data, which are growing by orders of magnitude in volume in remarkably short time periods.

It is this latter problem that NASA's Jet Propulsion Laboratory (JPL) began facing several years ago. Until recently, JPL would disseminate data collected by various instruments (Earth-based, orbiting, and in outer space) to the interested scientists around the United States by "burning" CD-ROMs and mailing them via the U.S. Postal Service. In addition to being slow, sequential, unidirectional, and lacking interactivity, this method was expensive, costing hundreds of thousands of dollars. Furthermore, the method was prone to security breaches, and the exact data distribution (determining which data goes to which destinations) had to be calculated for each individual shipment. It had become increasingly difficult to manage this process as the number of projects and missions, as well as involved scientists, grew. An even more critical limiting factor became the sheer volume of data that the current (e.g., Planetary Data System, or PDS), pending (e.g., Mars Reconnaissance Orbiter, or MRO), and planned (e.g., Lunar Reconnaissance Orbiter, or LRO) missions would produce: from terabytes (PDS), to hundreds of terabytes (MRO), to petabytes or more (LRO). Clearly, spending millions of dollars *just* to distribute the data to scientists is impractical.

This prompted NASA's Office of Space Science to explore construction of an end-to-end software framework that would lower the cost of distributing and managing scientific data, from the inception of data at a science processing center to its ultimate arrival on the desks of interested users. Because of increasing data volumes, the framework had to be *scalable* and have native support for *evolution* to hundreds of sites and thousands of data types. Additionally, the framework had to enable the virtualization of *heterogeneous* data (and processing) sources, and to address wide-scale (national and international) *distribution* of data. The framework needed to be *flexible*: it needed to support fully automated processing of data throughout its lifecycle, while still allowing interactivity and intervention from an operator when needed. Furthermore because data is itself distributed across NASA agencies, any software framework that distributes NASA's data would require the capability for *tailorable* levels of *security* and for varying *types of users* belonging to *multiple organizations*.

There were also miscellaneous issues of data ownership that needed to be overcome. Ultimately, because NASA's science data is so distributed, the owners of data systems (e.g., a Planetary

ICSE06', May 20–28, 2006, Shanghai, China.

Science Principal Investigator) feel hard pressed to *control* their data, as the successful operation and maintenance of their data systems are essential services that they provide. As such, any framework that virtualizes science data sources across NASA should be *transparent* and *unobtrusive*: it should enable dissemination and retrieval of data across data systems, each of which may have their own external interfaces and services; at the same time, it should enable scientists to maintain and operate their data systems *independently*. Finally, to lower costs, once the framework was built and installed, it needed to be *reusable*, *free*, and *distributable* to other NASA sites and centers for use.

Over the past seven years we have designed, implemented and deployed a framework called OODT (Object Oriented Data Technology) that has met these rigorous demands. In this paper we discuss the significant software engineering challenges we faced in developing OODT. The primary objective of the paper is to demonstrate how OODT's explicit software architectural basis enabled us to effectively address these challenges. In particular, we will detail the architectural decisions we found most difficult and/or critical to OODT's ultimate success. We highlight several representative examples of OODT's use to date both at NASA and externally. We contrast our solution with related approaches, and argue that a major differentiator of this work, in addition to its explicit architectural foundation, is its native support for architecture-based development of distributed scientific applications.

2. SOFTWARE ENGINEERING CHALLENGES

To develop OODT, we needed to address several significant software engineering challenges, the bulk of which surfaced in light of the complex data management and distribution issues regularly faced within a distributed, large-scale government organization such as NASA. In this paper we will focus on nine key challenges: *Complexity*, *Heterogeneity*, *Location Transparency*, *Autonomy*, *Dynamism*, *Scalability*, *Distribution*, *Decentralization*, and *Performance*.

Complexity – We envisioned OODT to be a large, multi-site, multi-user, complex system. At the software level, complexity ranged from understanding how to install, integrate, and manage the software remotely deployed at participating organizations, to understanding how to manage information such as access privileges and security credentials across both NASA and non-NASA sites. There were also complexities at the software networking layer, including varying firewall capabilities at each institution, and data repositories that would periodically go offline and needed to be remotely restarted. Just understanding the varying types of data held at sites linked together via OODT was a significant task. Even sites within the same science domain (e.g., planetary science) describe similar data sets in decidedly different ways. Discerning in what ways these different data models were common and what attributes of data could be shared, done away with, or amended, was a huge challenge. Finally, the different interfaces to data, ranging from third-party, well-engineered database management systems, to in-house data systems, ultimately to flat text file-based data was a particularly difficult challenge that we had to hurdle.

Heterogeneity – In order to drive down the data management costs for science missions, the same OODT framework needed to span multiple science domains. The domains initially targeted were earth and planetary; this has subsequently been expanded to space, biomedical sciences, and the modeling and simulation communities. As such, the same core set of OODT software components, system designs, and implementation-level facilities had to work across widely varying science domains.

The data management processes within the organizations that use OODT also added to its heterogeneity. For instance, OODT components needed to have interfaces with end users and support interactive sessions, but also with scientific instruments, which most likely were automatic and non-interactive. Scientific instruments could push data to certain components in OODT, while other OODT components would need to distribute data to users outside of OODT. End-users in some cases wanted to perform transformations on the data sent to them by OODT, and then to return the data back into OODT. The framework needed to support scenarios such as these seamlessly.

Many other constraints also imposed the heterogeneity requirement on OODT. We can group these constraints into two major categories:

- *Organizational* – As we briefly alluded above, discipline experts who wanted to disseminate their data via OODT really wanted the data to reside at their respective institutions. This constraint non-negotiable, and significantly impacted the space of technical solutions that we could investigate for OODT.
- *Technical* – Since OODT had to federate many different data holdings and catalogs, we faced the constraints of linking them together and federating very different schemas and varying levels of sophistication in the data system interfaces (e.g., flat files, DBMS, web pages). Even those systems managing data through "higher level APIs" and middleware (e.g., RMI, CORBA, SOAP) proved non-trivial to integrate.

The constraints enjoined by heterogeneity alone led us to realize that the OODT framework would need to draw heavily from multiple areas. Database systems, although used successfully for many years to manage large amounts of data at many sites, lacked the flexibility and interface capability to integrate data from other more crude APIs and storage systems (such as a PI-led web site). Databases also did not address the distribution of data and "ownership" issues. The advent of the web, although a promising means for providing openness and flexible interfaces to data, would not alone address the issues such as multi-institutional security and access. Furthermore, its request/reply nature would not easily handle other distribution scenarios, e.g., subscribe/notify. Research in the area of grid computing [1] has defined "out of the box" services for managing data systems (e.g., GridFTP), but which utilized alone would not address our other challenges (e.g., complexity).

Location Transparency – Even though data could potentially be input into and output from the system from many geographically disparate and distributed sites, it should appear to the end-users as if the data flow occurred from a single location. This requirement was reinforced by the need to dynamically add data producers and consumers to a system supported by OODT, as will be further discussed below.

Autonomy – When designing the OODT framework, we could not dictate how data providers should store, process, find, evolve, or retire their data. Instead, the framework needed to be

transparent, allowing data providers to continue with their regular business processes, while managing and disseminating their information unobtrusively.

Dynamism – It is expected that data providers for the most part will be stable organizations. However, there are cases in which new data producing (occasionally) and consuming (frequently) nodes will need to be brought on-line. Back-end data sources need to be pluggable, with little or no direct impact on the end-user of the OODT system, or on the organization that owns the data source. New end-users (or client hosts) should also be able to "come and go" without any disruption to the rest of the system. In the end, we realized this meant the whole infrastructure must be capable of some level of dynamism in order to meet these constraints.

Scalability – OODT needed to manage large volumes of data, from at least hundreds of gigabytes at its inception to the current missions which will produce hundreds of terabytes. The framework needed to support at least dozens of institutional data providers (which themselves may have subordinate data system providers), dozens of user types (e.g., scientists, teachers, students, policy makers), thousands of users, hundreds of geographic sites, and thousands of different data types to manage and disseminate.

Distribution – The framework should be able to handle the physical distribution of data across sites nationally and internationally, and ultimately the physical distribution of the system interfaces which provide the data.

Decentralization – Each site may have its own data management processes, interfaces and data types, which were operating independently for some time. We needed to devise a way of coordinating and managing data between these data sites and providers without centralizing control of their systems, or information. In other words, the requirement was that the different sites retain their full autonomy, and that OODT adapts instead.

Performance – Despite its scale and interaction with many organizations, data systems, and providers, OODT still needed to perform under stringent demands. Queries for information needed to be serviced quickly: in many cases response time under five seconds was used as a baseline. Additionally, OODT needed to be operational whenever any of the participating scientists wanted to locate, access, or process their data.

3. BACKGROUND AND RELATED WORK

Several large-scale software technologies that distribute, manage, and process information have been constructed over the past decade. Each of these technologies falls into one or more of four distinct areas: *grid-computing*, *information integration*, *databases*, and *middleware*. In this section, we briefly survey related projects in each of these areas and compare their foci and accomplishments to those of OODT. Additionally, since a major focal point of OODT is software architecture, we start out by providing some brief software architecture background and terminology to set the context.

Traditionally, *software architecture* has referred to the abstraction of a software system into its fundamental building blocks: software *components*, their methods of interaction (or software *connectors*), and the governing rules that guide the composition of software components and software connectors (*configurations*) [2, 3]. Software architecture has been recognized in many ways to be the *linchpin* of the software development process. Ideally, the software requirements are reflected within the software system's components and interactions; the components and interactions are captured within the system's architecture; and the architecture is used to guide the design, implementation, and evolution of the system. Design guidelines that have been proven effective are often codified into *architectural styles*, while specific architectural solutions (e.g., concrete system structures, component types and interfaces, and interaction facilities) within specific domains are captured as reusable *reference architectures*.

Grid computing deals with highly complex and distributed computational problems and large volume data management tasks. Massive parallel computation, distributed workflow, and petabyte scale data distribution are only a small cross-section of the grid's capabilities. Grid projects are usually broken down into two areas. *Computational* grid systems are concerned with solving complex scientific problems involving supercomputing scale resources dispersed across various organizational boundaries. The representative computational grid system is the Globus Toolkit [4]. Globus is built on top of a web-services [5] substrate and provides resource management components, distributed workflow and security infrastructure. Other computational grid systems provide similar capabilities. For example, Alchemi [6] is a .NET-based grid technology that supports distributed job scheduling and an object-oriented grid development environment. JCGrid [7] is a light weight, Java-based open source computational grid project whose goal is to support distributed job scheduling and the splitting of CPU-intensive tasks across multiple machines.

The other class of grid systems, *Data* grids, is involved in the management, processing, and distribution of large data volumes to disbursed and heterogeneous users, user types, and geographic locations. There are several major data grid projects. The LHC Computing Grid [8] is a system whose main goal is to provide a data management and processing infrastructure for the high energy physics community. The Earth System Grid [9] is geared towards supporting climate modeling research and distribution of climate data sets and metadata to the climate and weather scientific community.

Two independently conducted studies [10, 11] have identified three key areas that the current grid implementations must address more effectively in order to promote data and software interoperability: (1) formality in grid requirements specification, (2) rigorous architectural description, and (3) interoperability between grid solutions. As we will discuss in this paper, our work to date on OODT has the potential to be a stepping stone in each of these areas: its explicit focus on architectures for data-intensive, "grid-like" systems naturally addresses the three concerns.

There have been several well-known efforts within the AI and database communities that have delved into the topic of *information integration*, or the shared access, search, and retrieval of distributed, heterogeneous information resources. Within the past decade, there has been significant interest in building information mediators that can integrate information from multiple data sources. Mediators federate information by querying multiple data sources, and fusing back the gathered results. The representative systems using this approach include TSIMMS [12],

Information Manifold [13], The Internet Softbot [14], InfoSleuth [15], Infomaster [16], DISCO [17], SIMS [18] and Ariadne [19]. Each of these approaches focuses on fundamental algorithmic components of information integration: (1) formulating expressive, efficient query languages (such as Theseus [20]) that query many heterogeneous data stores; (2) accurately and reliably describing both global, and source data models (e.g. the Global-as-view [12] and Local-as-view [21] approaches); (3) providing a means for global-to-source data model integration; and (4) improving queries and deciding which data sources to query (e.g. query reformulation [22] and query rewriting [22, 23]).

However, these algorithmic techniques fail to address the software engineering side of information integration. For instance, existing literature fails to answer questions such as, which of the components in the different systems' architectures are common; how can they be reused; which portions of their implementations are tied to (which) software components; which software connectors are the components using to interact; are the interaction mechanisms replaceable (e.g., can a client-server interaction in Ariadne become a peer-to-peer interaction); and so on. Additionally, none of the above related mediator systems have formalized a process for designing, implementing, deploying, and maintaining the software components belonging to each system.

Several *middleware* technologies such as CORBA, Enterprise Java Beans [24], Java RMI [25], and more recently SOAP and Web services [5] have been suggested as "silver bullets" that address the problem of integrating and utilizing heterogeneous software computing and data resources. Each of these technologies provides three basic services: (1) an implementation and composition framework for software components, possibly written in different languages but conforming to a specific middleware interface; (2) a naming registry used to locate components; and (3) a set of basic services such as (un-)marshalling of data, concurrency, distribution and security.

Although middleware is very useful "glue" that can connect software components written in different languages or deployed in heterogeneous environments, middleware technologies do not provide any "out of the box" services that deal with computing and data resource management across organizational boundaries and across computing environments at a national scale. These kinds of services usually have to be engineered into the middleware itself. We should note that in grid computing such services are explicitly called out and provided at a higher layer of abstraction. In fact, the combination of these higher-level grid services and an underlying middleware platform is typically referred to as a "grid technology" [11].

4. OODT ARCHITECTURE

OODT's architecture is a *reference architecture* that is intended to be instantiated and tailored for use across science domains and projects. The reference architecture comprises several components and connectors. A particular instance of this reference architecture, that of NASA's planetary data system (PDS) project, is shown in Figure 1. OODT is installed on a given host inside a "sandbox", and is aware of and interacts only with the designated external data sources outside its sandbox. OODT's

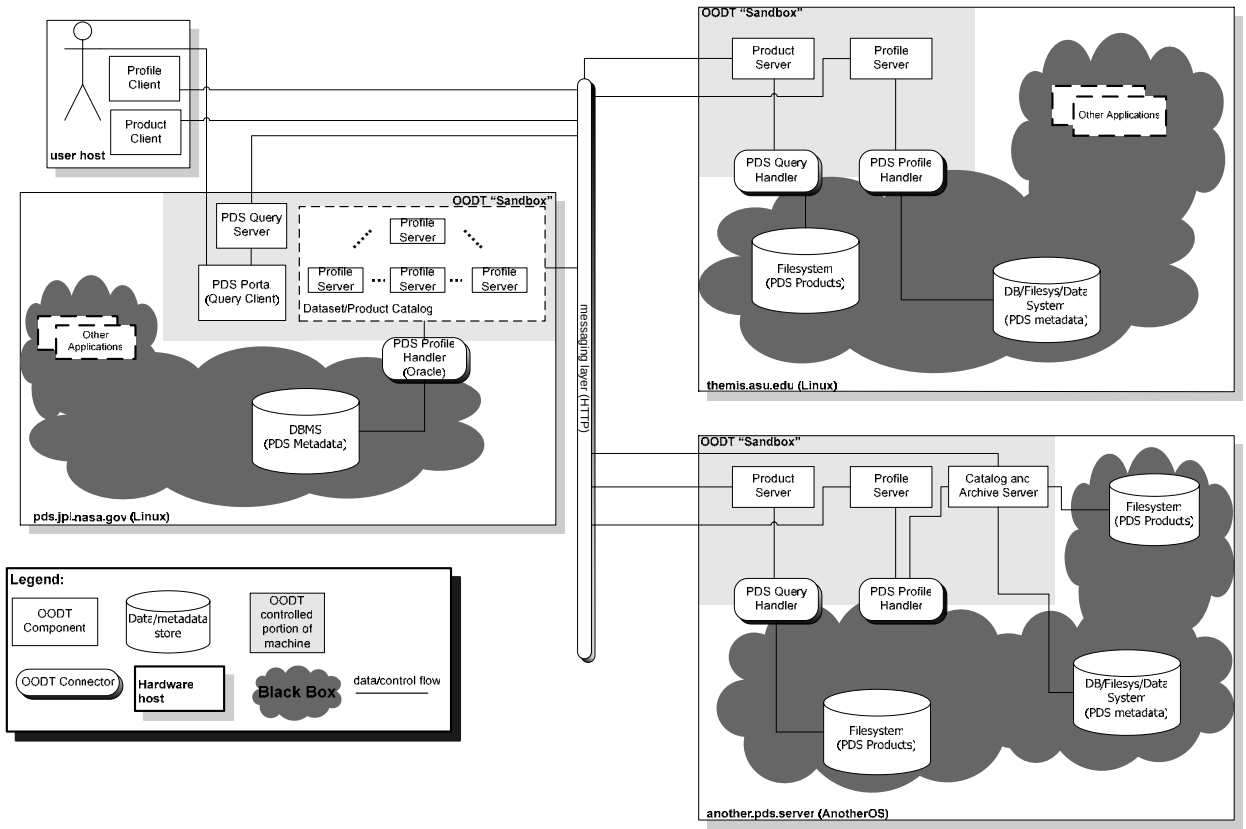

Figure 1. The Planetary Data System (PDS) OODT Architecture Instantiation

components are responsible for delivering data from heterogeneous data stores, identifying and locating data within the system, and ingesting and processing data into underlying data stores. The connectors are responsible for integrating OODT with heterogeneous data sources; providing reliable messaging to the software components; marshalling resource descriptions and transferring data between components; transactional communication between components; and security related issues such as identification, authorization, and authentication. In this section, we describe the guiding principles behind the reference architecture. We then describe each of the OODT reference components and connectors in detail. In Section 5, we describe specific instantiations of the reference architecture in the context of several projects that are using OODT.

4.1 Guiding Principles

The software engineering challenges discussed in Section 2 motivated and framed the development of OODT. Conquering these challenges led us to a set of four guiding principles behind the OODT reference architecture.

The first guiding principle is *division of labor*. Each capability provided by OODT (e.g., processing, ingestion, search, and retrieval of data, access to heterogeneous data, and so on) is carefully divided among separate, independent architectural components and connectors. As will be further detailed below, the principle is upheld through OODT's rigorous separation of concerns, and modularity enforced by explicit interfaces. This principle addresses the complexity, heterogeneity, dynamism, and decentralization challenges.

Closely related to the preceding principle is *technology independence*. This principle involves keeping up-to-date with the evolution of software technology (both in-house and third-party), while avoiding tying the OODT architecture to any specific implementation. By allowing us to select the technology most appropriate to a given task or specific need, this principle helps us to address the challenges of complexity, scalability, security, distribution, location transparency, performance, and dynamism. For instance, OODT's initial reference implementation used CORBA as the substrate for its messaging layer connector. When the CORBA vendor decided to begin charging JPL significant license fees (thus violating NASA's objective of producing a solution that would be free to its users), the principle of technology independence came into play. Because the OODT messaging layer connector supports a wrapper interface around the lower-level distribution technology, we were able to replace our initial CORBA-based connector with one using Java's open source RMI middleware, and redeploy the new connector to the OODT user sites, within three person days.

Another guiding principle of OODT is the distinguishing of *metadata as a first-class citizen* in the reference architecture, and separating metadata from data. The job of metadata (i.e., "data about data") is to describe the data universe in which the system is operating. Since OODT is meant to be a technology that integrates diverse data sources, this data universe is highly heterogeneous and possibly dynamic. Metadata in OODT is meant to catalog information, allowing a user to locate and describe the actual data in which she is interested. On the other hand, the job of *data* in OODT is to describe physical or scientific phenomena; it is the ultimate end user product that an OODT system should deliver. This principle helps to address the challenges of heterogeneity, autonomy of data providers, and decentralization.

Separating the data model from the software is another key principle behind the reference architecture. Akin to ontology/data-driven systems, OODT components should not be tied to the data and metadata that they manipulate. Instead, the components should be flexible enough to understand many (meta-)data models used across different scientific domains, without reengineering or tailoring of the component implementations. This principle helps to address the challenges of complexity and heterogeneity.

These four guiding principles are reified in a reference architecture comprising four pairs of component types and two classes of connectors organized in a canonical structure. One instantiation of the reference architecture reflecting the canonical structure is depicted in Figure 1. Each OODT architectural element (component and connector) serves a specific purpose, with its functionality exported through a well-defined interface. This supports OODT's constant evolution, allowing us to add, remove, and substitute, if necessary dynamically (i.e., at runtime), elements of a given type. It also allows us to introduce flexibility in the individual instances of the reference architecture while, at the same time, controlling the legal system configurations. Finally, the explicit connectors and well-defined component interfaces allow OODT in principle to integrate with a wide variety of third-party systems (e.g., [26]). The outcome of the guiding principles (described above) and design decisions (detailed below) is an architecture that is "easy to build, hard to break".

4.2 OODT Components

4.2.1 Product Server and Product Client

The *Product Server* is used to retrieve data from heterogeneous data stores. The product server accepts a query structure that identifies a set of zero or more products which should be returned the issuer of the query. A *product* is a unit of data in OODT and represents anything that a user of the system is interested in retrieving: a JPEG image of Mars, an MS Word document, a zip file containing text file results of a cancer study, and so on. Product servers can be located at remote data sites, geographically and/or institutionally disparate from other OODT components. Alternatively, product servers can be centralized, located at a single site. The objective of the product server is to deliver data from otherwise heterogeneous data stores and systems. As long as a data store (or system) provides some kind of access interface to get its data, a product server can "wrap" those interfaces with the help of *Handler* connectors described in Section 4.3 below.

The *Product Client* component communicates with a product server via the *Messaging Layer* connectors described in Section 4.3. A product client resides at the end-user's (e.g., scientist's) site. It must know the location of at least one product server, and the query structure that identifies the set of products that the user wants to retrieve. At the same time, it is completely insulated from any changes in the physical location or actual representation of the data; its only interface is to the product server(s). Many product clients may communicate with the same product server, and many product servers can return data to the same product client. This adds flexibility to the architecture without introducing unwanted long-term dependencies: a product client can be added,

removed, or replaced with another one that depends on different product servers, without any effect on the rest of the architecture.

4.2.2 Profile Server and Profile Client

The *Profile Server* manages resource description information, i.e., metadata, in a system built with OODT. Resource description information is divided into three main categories:

- *Housekeeping Information* – Metadata such as *ID*, *Last Modified Date*, *Last Revised By*. This information is kept about the resource descriptions themselves and is used by the profile server to inventory and catalog resource descriptions. This is a fixed set of metadata.
- *Resource Information* – This includes metadata such as *Title*, *Author*, *Creator*, *Publisher*, *Resource Type*, and *Resource Location*. This information is kept for all the data in the system, and is an extended version of the Dublin Core Metadata for describing electronic resources [27]. This is also a fixed set of metadata.
- *Domain-Specific Information* – This includes metadata specific to a particular data domain. For instance, in a cancer research system this may include metadata such as *Blood Specimen Type*, *Site ID*, and *Protocol/Study Description*. This set of metadata is flexible and is expected to change.

As with product servers, profile servers can be decentralized at multiple sites or centralized at a single site. The objective of the profile server is to deliver metadata that gives a user enough information to locate the actual data within OODT regardless of the underlying system's exact configuration, and degrees of complexity and heterogeneity; the user then retrieves the data via one or more product servers. Because profile servers do not serve the actual data, they need not have a direct interface to the data that they describe. In addition to the complete separation of duties between profile and product servers, this ensures their location independence, allows their separate evolution, and minimizes the effects of component and/or network failures in an OODT system.

Profile Client components communicate with profile servers over the messaging layer connectors. The client must know the location of the profile server, and must provide a query that identifies the metadata that a user is interested in retrieving. There can be many profile clients speaking with a single profile server, and many profile servers speaking with a single profile client. The architectural effects are analogous to those in the case of product clients and servers.

4.2.3 Query Server and Query Client

The *Query Server* component provides an integrated search and retrieval capability for the OODT reference architecture. Query servers interact with profile and product servers to retrieve metadata and data requested by system users. A query server is seeded with an initial set of references to profile servers. Upon receiving a query from a user, the query server passes it along to each profile server from its list, and collects the metadata returned. Part of this metadata is a *resource location* (recall Section 4.2.2) in the form of a URI [28]. A URI can be a link to a product server, to a web site with the actual data, or to some external data providing system. This directly supports heterogeneity, location transparency, and autonomy of data providers in OODT.

Another novel aspect of OODT's architecture is that if a profile server is unable to service the query, or if it believes that other profile servers it is aware of may contain relevant metadata, it will return the URIs of those profile servers; the query server may then forward the query to them. As a result, query servers are completely decoupled from product servers (and from any "exposed" external data sources), and are also decoupled from most of the profile servers. In turn, this lessens the complexity of implementing, integrating, and evolving query servers. Once the resource metadata is returned, the query server will either allow the user herself to use the supplied URIs to find the data in which she was interested (interactive mode), or it will retrieve, package, and deliver the data to the user (non-interactive mode). As with the product and profile servers, query servers can be centrally located at a single site, or they can be decentralized across multiple sites.

Query Client components communicate with the query servers. The query client must provide a query server with a query that identifies the data in which the user is interested, and it must set a mode for the query server (interactive or non-interactive mode). The query client may know the location of the query server that it wants to contact, or it may rely on the messaging layer connector to route its queries to one or more query servers.

4.2.4 Catalog and Archive Server and Client

The *Catalog and Archive Server (CAS)* component in OODT is responsible for providing a common mechanism for ingestion of data into a data store, including any processing required as a result of ingestion. For instance, prior to the ingestion of a poor-resolution image of Mars, the image may need to be refined and the resolution improved. CAS would handle this type of processing. Any data ingested into CAS must include associated metadata information so that the data can be cataloged for search and retrieval purposes. Upon ingestion, the data is sent to a data store for preservation, and the corresponding metadata is sent to the associated catalog. The data store and catalog need not be located on the same host; they may be located on remote sites provided there is an access mechanism to store and retrieve data from each. The goal of CAS is to streamline and standardize the process of adding data to an OODT-aware system. Note that a system whose data stores were populated prior to its integration into OODT can still use CAS for its new data. Since the CAS component populates data stores and catalogs with both data and metadata, specialized product and profile server components have been developed to serve data and metadata from the CAS backend data stores and catalogs more efficiently. Any older data can still be served with existing product and profile servers.

The *Archive Client* component communicates with CAS. The archive client must know the location of the CAS component, and must provide it with data to ingest. Many archive clients can communicate with a single CAS component, and vice versa. Both the archive client and CAS components are completely independent of the preceding three pairs of component types in the OODT reference architecture.

4.3 OODT Connectors

4.3.1 Handler Connectors

Handler connectors are responsible for enabling the interaction between OODT's components and third-party data stores. A handler connector performs the transformation between an underlying (meta-)data store's internal API for retrieving data and its (meta-)data format on the one hand, and the OODT system

on the other. Each handler connector is typically developed for a class of data stores and metadata systems. For example, for a given DBMS such as Oracle, and a given internal representation schema for metadata, a generic Oracle handler connector is typically developed and then reused. Similarly, for a given filesystem scheme for storing data, a generic filesystem handler connector is developed and reused across like filesystem data stores.

Each profile server and product server relies on one or more handler connectors. Profile servers use *profile handlers*, and product servers use *query handlers*. Handler connectors thereby completely insulate product and profile servers from the third-party data stores. Handlers also allow for different types of transformations on (meta-)data to be introduced dynamically without any effect on the rest of OODT components. For example, a product server that distributes Mars image data might be serviced by a query handler connector that returns high-resolution (e.g., 10 GB) JPEG image files of the latest summit climbed by a Mars rover; if the system ends up experiencing performance problems, another handler may be (temporarily) added to return lower-resolution (e.g., 1 MB) JPEG image files of the same scenario. Likewise, a profile server may have two profile handler connectors, one that returns image-quality metadata (e.g., *resolution* and *bits/pixel*) and another that returns instrument metadata about Mars rover images (e.g., *instrument name* or *image creation date*).

4.3.2 Messaging Layer Connector

The *Messaging Layer* connector is responsible for marshalling data and metadata between components in an OODT system. The messaging layer must keep track of the locations of the components, what types of components reside in which locations, and if components are still running or not. Additionally, the messaging layer is responsible for taking care of any needed security mechanisms such as authentication against an LDAP directory service, or authorization of a user to perform certain role-based actions.

The messaging layer in OODT provides synchronous interaction among the components, and some delivery guarantees on messages transferred between the software components. Typically in any large-scale data system, the asynchronous mode of interaction is not encouraged because partial data transfers are of no use to users such as scientists who need to make analysis on entire data sets.

The messaging layer supports communication between any number of connected OODT software components. In addition, the messaging layer natively supports connections to other messaging layer connectors as well. This provides us with the ability to extend and adapt an OODT system's architecture, as well as easily tailor the architecture for any specific interaction needs (e.g., by adding data encryption and/or compression capabilities to the connector).

5. EXPERIENCE AND CASE STUDIES

The OODT framework has been used both within and outside NASA. JPL, NASA's Ames Research Center, the National Institutes of Health (NIH), the National Cancer Institute (NCI), several research universities, and U.S. Federally Funded Research and Development Centers (FFRDCs) are all using OODT in some form or fashion. OODT is also available for download through a large open-source software distributor [29]. OODT components are found in planetary science, earth science, biomedical, and clinical research projects. In this section, we discuss our experience with OODT in several representative projects within these scientific areas. We compare and contrast how the projects were handled before and after OODT. We sketch some of the domain-specific technical challenges we encountered and identify how OODT helped to solve them.

To begin using OODT, a user designs a deployment architecture from one or more of the reference OODT components (e.g., product and profile servers), and the reference OODT connectors. The user must determine if any existing handler connectors can be reused, or if specialized handler connectors need to be developed. Once all the components are ready, the user has two options for deploying her architecture to the target hosts: (1) the user may translate her design into a specialized OODT deployment descriptor XML file, which can then be used to start each program on the target host(s); or (2) the user can deploy her OODT architecture using a remote server control component, adding components, and connectors via a graphical user interface. The GUI allows the user to send component and connector code to the target hosts, to start, shut-down, and restart the components and connectors, and to monitor their health during execution.

5.1 Planetary Data System

One of the flagship deployments of OODT has been for NASA's Planetary Data System (PDS) [30]. PDS consists of seven "discipline nodes" and an engineering and management node. Each node resides at a different U.S. university or government agency, and is managed autonomously.

For many years PDS distributed its data and metadata on physical media, primarily CD-ROM. Each CD-ROM was formatted a according to a "home-grown" directory layout structure called an *archive volume*, which later was turned into a PDS standard. PDS metadata was constructed using a common, well-structured set of 1200 metadata elements, such as *Target Name* and *Instrument Type*, that were identified from the onset of the PDS project by planetary scientists. Beginning in the late 1990s the advent of the WWW and the increasing data volumes of missions led NASA managers to impose a new paradigm for distributing data to the users of the PDS: data and metadata were now to be distributed electronically, via a single, unified web portal. The web portal and accompanying infrastructure to distribute PDS data and metadata was built in 2001 using OODT in the manner depicted in Figure 1.

We faced several technical challenges deploying OODT to PDS. PDS data and metadata were highly distributed, spanning all seven of the scientific discipline nodes across the country. Although the entire data volume across PDS at the time was around 7 terabytes, it was estimated that the volume would grow to 10 terabytes by 2004. Consequently, the system needed to be scalable and respond to large growth spurts caused by new data producing missions. The flexibility and modularity of the OODT product and profile server components were particularly useful in this regard. Using a product and/or profile server, each new data producing system in the PDS could be dynamically "plugged in" to the existing PDS infrastructure that we constructed, without disturbing existing components and processes.

We also faced the problem of heterogeneity. Almost every node within PDS had a different operating system, ranging from Linux, to Windows, to Solaris, to Mac OS X. Each node

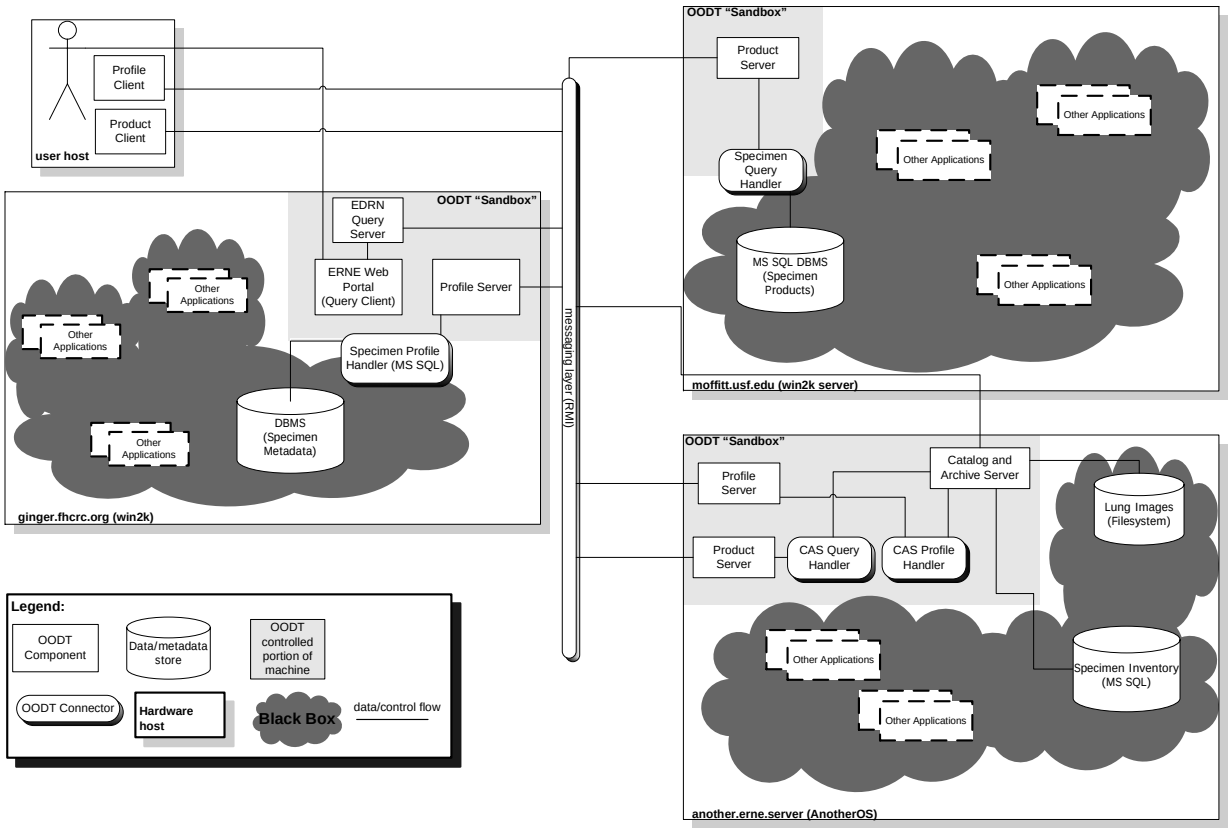

Figure 2. The Early Detection Research Network (EDRN) OODT Architecture Instantiation

maintained its own local catalog system. Although each node in PDS had different file system implementations dictated by their OS, each node stored their data and metadata according to the archive volume structure. Because of this, we were able to write a single, reusable *PDS Query Handler* which could serve back products from a PDS archive volume structure located on a file system. Plugging into each node's catalog system proved to be a significant challenge. For nearly all of the nodes, specialized profile handler connectors were constructed to interface with the underlying catalog systems, which ranged from static text files called *PDS label files* to dynamic web site inventory systems constructed using Java Server Pages. Because each of the catalogs tagged PDS data using the common set of 1200 elements, we were able to share much of the code base among the profile handler connectors, ultimately only changing the portion of the code that made the particular JSP page call, or read the selected set of metadata from the label file. The entire code base of the PDS including all the domain specific handler connectors is only slightly over 15 KSLOC, illustrating the high degree of reusability provided by the OODT framework.

5.2 Early Detection Research Network

OODT is also supporting the National Cancer Institute's (NCI) Early Detection Research Network (EDRN). EDRN is a distributed research program that unites researchers from over thirty institutions across the United States. Tens of thousands of scientists participate in the EDRN. Each institution is focused on the discovery of cancer biomarkers as indicators for disease [31]. A critical need for the EDRN is an electronic infrastructure to support discovery and validation of these markers.

In 2001 we worked with the EDRN program to develop the first component of their electronic biomarker infrastructure called the EDRN Resource Network Exchange (ERNE). The (partial) corresponding architecture is depicted in Figure 2. One of the major goals of ERNE was to provide real-time access to bio-specimen information across the institutions of the EDRN. Bio-specimen information typically consisted of gigabytes of specimen images, and location and contact metadata for obtaining the specimen from its origin study institution. The previous method of obtaining bio-specimen information was very human-intensive: it involved phone calls and some forms of electronic communication such as email. Specimen information was not searchable across institutions participating in the EDRN. The bio-specimen catalogs were largely out-of-date, and out-of-synch with current holdings at each participating institution.

One of the initial technical challenges we faced with EDRN was scale. The EDRN was over three times as large as the PDS. Because of this we chose to target ten institutions initially, rather than the entire set of thirty one. Again, OODT's modularity and scalability came into play as we could phase deployment at each deployment institution. As we instantiated new product, profile, query, and archive servers at each institution, we could do so without interrupting any existing OODT infrastructure already deployed.

Another challenge that we encountered was dealing with each participating site's Institutional Review Board (IRB). An IRB is required to review and ensure compliance of projects with

federal laws related to working with data from research projects involving human subjects. To satisfy the IRB, any OODT components deployed at an EDRN site had to provide an adequate security capability in order to get approval to share the data externally from an institution. OODT's separation of data and metadata explicitly allowed us to satisfy this requirement. We designed ERNE so that each institution could remain in control of their specimen holding data by instantiating product server components at each site, rather than distributing the information across ERNE which would have violated the IRB agreements.

Another significant challenge we faced in developing ERNE was lack of a consistent metadata model for each ERNE site. We were forced to develop a common specimen metadata model and then to create specific mappings to link each local site to the common model. OODT aided us once again in this endeavor as the common mappings we developed were easily codified into a query handler connector, and reused across each ERNE site.

The entire code base of ERNE, including all its specialized handler connectors is only slightly over 5.3 KSLOC, highlighting the high degree of reusability of the shared framework code base and the handler code base.

5.3 Science Processing Systems

OODT has also been deployed in several science processing system missions both, operational and under development. Due to space limitations, we can only briefly summarize each of the OODT deployments in these systems.

SeaWinds, a NASA-funded earth science instrument flying on the Japanese ADEOS-II spacecraft, used the OODT CAS component as a workflow and processing component for its Processing and Analysis Center (SeaPAC). SeaWinds produced several gigabytes of data during its six year mission. CAS was used to control the execution and data flow of mission-specific data processor components, which calibrated and created derived data products from raw instrument data, and archived those products for distribution into the data store managed by CAS. A major challenge we faced during the development of SeaPAC was that the processor components were developed by a group outside of the SeaWinds project. We had to provide a mechanism for integrating their source code into the OODT SeaPAC framework. OODT's separation of concerns allowed us to address this issue with relative ease: once the data processors were finished, we were able wrap and tailor them internally within CAS, without disturbing the existing SeaPaC infrastructure.

The success of the CAS within SeaWinds led to its reuse on several different missions. Another earth science mission called QuikSCAT retrofitted and replaced some of their existing processing components with CAS, using the SeaWinds experience as an example. The Orbiting Carbon Observatory (OCO) mission that will fly in 2009, and that is currently under development, is also utilizing CAS to ingest and process existing FTS CO^2 spectrometer data from earth-based instruments. The James Web Telescope (JWT) is using the CAS for to implement its workflow and processing capabilities for astrophysics data and metadata. Each of these science processing systems will face similar technical challenges, including separation of concerns between the actual processing framework and the developers writing the processor code, the volume of data that must be handled by the processing system (OCO is projected to produce over 150 *tera*bytes), and the flexibility and tailorability of the workflow needed to process the data. We believe that OODT is uniquely positioned to address these difficult challenges.

5.4 Computer Modeling Simulation and Visualization

OODT has also been deployed to aid the Computer Modeling Simulation and Visualization (CMSV) community at JPL, by linking together several institutional model repositories across the organizations within the lab, and creating a web portal interface to query the integrated model repositories. We developed specialized profile server components that locate and link to different model resources across JPL, such as power subsystem models of the Mars Exploration Rovers (MER), CAD-drawing models of different spacecraft assembly parts, and systems architecture models for engineering and design of spacecraft. Each of these different model types lived in separate independent repositories across JPL. For instance, the CAD models were stored in a commercial product called TeamCenter Enterprise [32], while the power and systems architecture models were stored in a commercial product called Xerox Docushare [33].

To integrate these model repositories for CMSV, we had to derive a common set of metadata across the wide spectrum of different model types that existed at JPL. OODT's separation of data from metadata allowed us to rapidly instantiate our common metadata model once we developed it, by constructing specialized profile handler connectors that mapped each repository's local model to the common model. Reusability levels were high across the connectors, resulting in an extremely small code base of 2.57 KSLOC.

Another challenge in light of this mapping activity was interfacing with the APIs of the underlying model repositories. In the above two cases, the APIs were commercial products, and poorly documented. In some cases, such as the Docushare repository, the APIs did not fully conform to their stated specifications. The division of labor amongst OODT components came into play on this task. It allowed us to focus on deploying the rest of the OODT supporting infrastructure, such as the web portal, and the profile handler connectors, and not getting stalled waiting for the support teams from each of the commercial vendors to debug our API problems. Once the OODT CMSV infrastructure was deployed, the modeling and simulation community at JPL immediately began adopting it and sharing their models across the lab. During the past year, the system has received around 40,000 hits on the web portal, and over 9,000 queries for models.

6. CONCLUSIONS

When the need arose at NASA seven years ago for a data distribution and management solution that satisfied the formidable requirements outlined in this paper, it was not clear to us initially how to approach the problem. On the surface, several applicable solutions already existed (middleware, information integration systems, and the emerging grid technologies). Adopting one of them seemed to be a preferable path because it would have saved us precious time. However, upon closer inspection we realized that each of these options could be instructive, but that none of them solved the problem we were facing (and that even some of these technologies themselves were facing).

The observation that directly inspired OODT was that we were dealing with *software engineering* challenges, and that those

challenges naturally required a software engineering solution. OODT is a large, complex, dynamic system, distributed across many sites, servicing many different users, and classes of users, with large amounts of heterogeneous data, possibly spanning multiple domains. Software engineering research and practice both suggest that success in developing such a system will be determined to a large extent by the system's *software architecture*. It therefore became imperative that we rely on our experience within the domain of data-intensive systems (e.g., JPL's PDS project), as well as our study of related research and practice, in order to develop an architecture for OODT that will address the challenges we discussed in Section 2. Once the architecture was designed and evaluated, OODT's initial implementation and its subsequent adaptations followed naturally.

As OODT's developers we are heartened, but as software engineering researchers and practitioners disappointed, that OODT still appears to be the only system of its kind. The intersection of middleware, information management, and grid computing is rapidly growing, yet it is still characterized by one-off solutions targeted at very specific problems in specific domains. Unfortunately, these solutions are sometimes clever by accident and more frequently little more than "hacks". We believe that OODT's approach is more appropriate, more effective, more broadly applicable, and certainly more helpful to developers of future systems in this area. We consider OODT's demonstrated ability to evolve and its applicability in a growing number of science domains to be a testament to its explicit, carefully crafted software architecture.

7. ACKNOWLEDGEMENTS

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

8. REFERENCES

[1] A. Chervenak, I. Foster, et al., "The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Data Sets," *J. of Network and Computer Applications*, vol. 23, pp. 187-200, 2000.

[2] N. Medvidovic and R. N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE TSE*, vol. 26, pp. 70-93, 2000.

[3] D. E. Perry and A. L. Wolf, "Foundations for the Study of Software Architecture," *Software Engineering Notes (SEN)*, vol. 17, pp. 40-52, 1992.

[4] "The Globus Alliance (http://www.globus.org)," 2005.

[5] "Webservices.org (http://www.webservices.org)," 2005.

[6] A. Luther, R. Buyya, et al., "Alchemi: A .NET-based Enterprise Grid Computing System," in *Proc. of 6th International Conference on Internet Computing*, Las Vegas, NV, USA, 2005.

[7] "JCGrid Web Site (http://jcgrid.sourceforge.net)," 2005.

[8] "LHC Computing Grid (http://lcg.web.cern.ch/LCG/)," 2005.

[9] D. Bernholdt, S. Bharathi, et al., "The Earth System Grid: Supporting the Next Generation of Climate Modeling Research," *Proceedings of the IEEE*, vol. 93, pp. 485-495, 2005.

[10] A. Finkelstein, C. Gryce, et al., "Relating Requirements and Architectures: A Study of Data Grids," *J. of Grid Computing*, vol. 2, pp. 207-222, 2004.

[11] C. A. Mattmann, N. Medvidovic, et al., "Unlocking the Grid," in *Proc. of CBSE*, St. Louis, MO, pp. 322-336, 2005.

[12] J. Hammer, H. Garcia-Molina, et al., "Information translation, mediation, and mosaic-based browsing in the tsimmis system," in *Proc. of ACM SIGMOD International Conference on Management of Data*, San Jose, CA, pp. 483-487, 1995.

[13] T. Kirk, A. Y. Levy, et al., "The information manifold," *Working Notes of the AAAI Spring Symposium on Information Gathering in Heterogeneous, Distributed Environment*, Menlo Park, CA, Technical Report SS-95-08, 1995.

[14] O. Etzioni and D. S. Weld, "A softbot-based interface to the Internet," *CACM*, vol. 37, pp. 72-76, 1994.

[15] A. Goñi, A. Illarramendi, et al., "An optimal cache for a federated database system," *Journal of Intelligent Information Systems*, vol. 9, pp. 125-155, 1997.

[16] M. R. Genesereth, A. Keller, et al., "Infomaster: An information integration system," in *Proc. of ACM SIGMOD International Conference on Management of Data*, Tucson, AZ, pp. 539-542, 1997.

[17] A. Tomasic, L. Raschid, et al., "A data model and query processing techniques for scaling access to distributed heterogeneous databases in disco," *IEEE Transactions on Computers*, 1997.

[18] Y. Arens, C. A. Knoblock, et al., "Query Reformulation for Dynamic Information Integration," *Journal of Intelligent Information Systems*, vol. 6, pp. 99-130, 1996.

[19] J. Ambite, N. Ashish, et al., "Ariadne: A system for constructing mediators for internet sources," in *Proc. of ACM SIGMOD International Conference on Management of Data*, Seattle, WA, pp. 561-563, 1998.

[20] G. Barish and C. A. Knoblock, "An Expressive and Efficient Language for Information Gathering on the Web," in *Proc. of 6th International Conference on AI Planning and Scheduling (AIPS-2002) Workshop*, Toulouse, France, 2002.

[21] A. Y. Halevy, "Answering queries using views: A survey," *VLDB Journal*, vol. 10, pp. 270-294, 2001.

[22] J. L. Ambite, C. A. Knoblock, et al., "Compiling Source Descriptions for Efficient and Flexible Information Integration," *Information Systems Journal*, vol. 16, pp. 149-187, 2001.

[23] E. Lambrecht and S. Kambhampati, "Planning for Information Gathering: A Tutorial Survey," ASU CSE Technical Report 96-017, May 1997.

[24] "Enterprise Java Beans (http://java.sun.com/ejb)," pp. 2005.

[25] "Java RMI (http://java.sun.com/rmi/)," 2005.

[26] C. A. Mattmann, S. Malek, et al., "GLIDE: A Grid-based Lightweight Infrastructure for Data-intensive Environments," in *Proc. of European Grid Conference*, Amsterdam, the Netherlands, pp. 68-77, 2005.

[27] DCMI, "Dublin Core Metadata Element Set," 1999.

[28] T. Berners-Lee, R. Fielding, et al., "Uniform Resource Identifiers (URI): Generic Syntax," 1998.

[29] "Open Channel Foundation: Request Object Oriented Data Technology (OODT) - (http://openchannelsoftware.com/orders/index.php?group_id=332)," 2005.

[30] J. S. Hughes and S. K. McMahon, "The Planetary Data System. A Case Study in the Development and Management of Meta-Data for a Scientific Digital Library.," in *Proc. of ECDL*, pp. 335-350, 1998.

[31] S. Srivastava, *Informatics in proteomics*. Boca Raton, FL: Taylor & Francis/CRC Press, 2005.

[32] "UGS Products: TeamCenter (http://www.ugs.com/products/teamcenter/)," 2005.

[33] "Document Management | Xerox Docushre (http://docushare.xerox.com/ds/)," 2005.