Software Architecture for Large-Scale, Distributed, Data-Intensive Systems

Chris A. Mattmann, Nenad Medvidovic and Paul M. Ramirez
University of Southern California
Computer Science Department
{mattmann,neno,pmramire}@usc.edu

Daniel J. Crichton, J. Steven Hughes and Sean C. Kelly
Jet Propulsion Laboratory
4800 Oak Grove Drive
{dcrichton,jshughes,sean.kelly}@jpl.nasa.gov

Abstract

The sheer amount of data produced by modern science research has created a need for the construction and understanding of "data-intensive systems", large-scale, distributed systems which are IO-bound (Moore et al. 1999). The formal nature of constructing such software systems; however, is relatively unstudied, and has been a large focus of the super-computing and distributed computing communities, rather than the software engineering communities. These data-intensive systems exhibit characteristics which appear fruitful for research from a software engineering, and software architectural focus. From our experience, the methodologies and formal notations for design and implementation of data-intensive systems look to be a good starting point for this important research area. This paper presents our experience with OODT, a novel software architectural style, and middleware-based implementation for data-intensive systems. To date, OODT has been successfully evaluated in several different science domains including Cancer Research with the National Cancer Institute (NCI), and Planetary Science with NASA's Planetary Data System (PDS).

1. Introduction

With the advent of new technologies and paradigms such as Grid-based systems (Globus 2004), distributed-object middlewares (Dashofy & Taylor & Medvidovic 1999) and wrapper-based information extraction tools (Knoblock et al 2000), there has been a change in practice from developing one-off data management solutions which manage data independently (with little attention paid to future interoperability with other systems), to developing architectures and middlewares which are able to integrate and *re-use* existing data *resources*, such as legacy data systems and legacy databases. This potential for re-use allows one to imagine and more importantly realize, the construction of large-scale distributed data management systems, whose main purpose is to *query, locate, access, process* and *distribute* data for potential users.

The *data-intensive* nature of these systems; however, shifts the focus from traditional software engineering methodologies and practices (which would involve *generating requirements*, converting those requirements into *architectural elements*, *implementing the system architecture*, and *evolving* the system as needed) to believing that hardware processing power, parallelism, and technology alone will be enough to construct software which:

1. Accesses data in legacy data resources
2. Discovers data which may not be known to the system initially at run-time
3. Correlates the different data models which describe each data resource
4. Correlates the software interfaces to the legacy data resources.

Existing work in these data-intensive systems severely lacks any common software engineering design constructs and relations between architectural components and implementation-level decisions. Our work has focused on this very area, the design and construction of large-scale, integrated data-intensive systems. We have developed an architectural style for data-intensive systems which allows software developers to construct architectural designs which satisfy data-intensive requirements (items 1-4 above), and middleware implementation framework of the OODT style software components and software connectors. Programmers can then take these designs, and use the middleware to implement the desired system functionality. We have coined our

architecture, and subsequent middleware implementation, OODT (*Object Orientated Data Technology*).

Section 2 describes the OODT architectural style. Section 3 covers the OODT middleware; a java based reference implementation of the OODT architectural components, and connectors, and ties the middleware back to the architectural style assumptions and properties. Section 4 presents two case studies of the OODT middleware deployment in two different science domains which share many of the same data-intensive system requirements presented above. Section 5 surveys related work in data-intensive systems (such as *Data Grids*) and architectural styles for large-scale, distributed data-intensive systems. Section 6 addresses open issues within OODT. Section 7 presents future research directions with software architecture for large-scale, data-intensive systems. Section 8 concludes our discussions.

2. An Architectural Style for Data-Intensive Systems

One method of defining an architectural style has been to define the primitive building blocks, software components and connectors, and then define legal topologies of these primitive building blocks needed to construct systems that implement the style itself (Taylor et al. 1996). Using these core definitions, system implementations are then constructed in the style, and analyzed to verify that key assumptions of the style at *design-time* are validated by the implemented system at *implementation time*. We use this method to construct the basis for the OODT architectural style.

2.1 Components

The core components for OODT are Product Servers, Profile Servers and Query Servers (Crichton & Hughes & Kelly 2002).

- **Product Servers** abstract away data resource interfaces (such as SQL, File Systems, HTTP) into system-independent interfaces for retrieval of data which satisfy the user queries.
- **Profile Servers** serve back scientific metadata (Crichton & Hughes & Kelly 2002) in the form of resource profiles, which provide methods for deciding what resources satisfy particular queries.
- **Query Servers** accept user queries, and then use profile servers to determine what resources to query and collect in order to satisfy the query and return data products. The Query Server then aggregates and returns back the federated data products to the caller.

2.1.1 Product Servers

Product Servers abstract away system dependent interfaces to data by wrapping (Sneed 1997) the system dependent interface such that it can support OODT queries. OODT queries are in the format of one or more (keyword logical operator value) predicates which themselves are joined by zero or more logical operators (such as AND and OR). More formally, a query q can be represented as:

$$(1) \quad q = \{(k_1, v_1, op_1)..\left[op_j(k_n, v_n, op_n)\right]\}$$

→Where k is the keyword, v is the value, and op is the logical operator

At the system interface level, OODT queries are posed against a set of *Common Data Elements* (which describe a high-level data model for the integrated data system, referred to as a Data Dictionary), and thus the set of allowable keywords k in each (keyword logical operator value) predicate comes from the set of the Common Data Elements in the system Data Dictionary. Each Product Server contains zero or more, *Query Handlers*. Because there may be multiple types of data products that are stored in the same data source, there may exist Query Handlers which serve back different products (a case like this could be imagined in a relational database, which stores both binary images of Mars, and also metadata containing information about the instrument which captured each image). Each Query Handler contains an abstract translation function t(q), which maps the query from the Common Data Element domain, to a query in the domain which the system dependent interface understands (i.e. maps Common Data Elements against *Source Data Elements*, Data Elements in the system-dependent domain). This function is represented by:

$$(2) \quad t(q) = q \in D_1, q \longrightarrow q', q' \in D_2$$

Essentially this function takes a query in Domain 1, and maps it to a new query, q', in domain 2. Because Product Server translation functions are implemented at implementation time, rather than design time, it is also possible for the implementer to define even more abstract translation functions which aren't described

here, but can be envisioned from the discussion above. The main goal is to take the abstract query against the Common Data Elements, and map that into something the underlying data system, or resource understands. Figure 2 depicts the internal structure of the Product Server component.

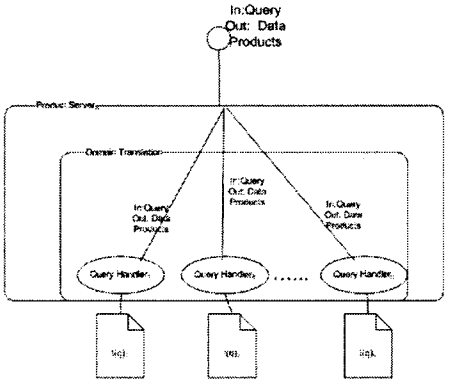

Figure 2. The internal structure of the Product Server component.

2.1.2 Profile Servers

Profile Servers respond to system queries, and return resources which can satisfy the system query. To achieve this functionality, Profile Servers store resource *Profiles* which describe the Data Elements, and semantic relationships that each data resource supports. In this fashion, a Profile Server can receive an OODT query, and match each keyword Data Element and associated semantic constraints, to a particular Resource that a profile describes, and return the matching profile(s) which satisfy the query. This scenario is illustrated in Figure 3.

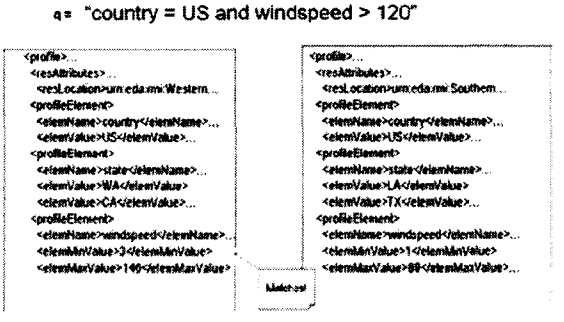

Figure 3. Query q, and matching profile of resource.

The profile structure is based on two internationally accepted standards, ISO/IEC 11179 (ISO 1999) to describe the structure of Data Elements, and the Dublin Core (DCMI 1999) set of Data Elements, which are used to describe any electronic resource. Our choices for these standards are explained further in (Crichton & Hughes & Kelly 2002) and beyond the scope of this paper, but we refer to them here in order to motivate the discussion of the profile structure. The profile contains metadata about the profile itself, such as a unique ID, a name for the profile, and an author of the profile. We refer to this metadata as *Profile Attributes*. The next piece of the metadata that the profile contains is an implementation and extension of the Dublin Core Elements to describe the resource that the profile describes, elements such as Creator, Name and Author. Further, our Dublin Core extensions include the addition of specific Data Elements to describe resources, including *Resource Location* and a taxonomy to categorize the type of resource (which we refer to as *Resource Class*). A Resource Location can be a Product Server, another Profile Server, or the Data Product itself. Resource Class is a '.' delimited character string such as `data.granule`, which implies that the resource described by the profile is a low-level Data Product. We refer to this extended Dublin Core metadata as *Resource Attributes*. The final piece of metadata that we include in profiles is domain-specific Data Elements which describe the resource in its originating domain. For example, in the Planetary Data System (Hughes & McMahon 1998), the Data Element "Target Name" is used to describe each data product's originating "target" (i.e. planet). These domain-specific Data Elements are called *Profile Elements*. The profile metadata is summarized in Table 1 below.

Table 1. Metadata stored in each resource profile.

	Profile Metadata	
Metadata	*Description of Metadata*	*Example*
Profile Attributes	Metadata describing the profile itself.	ProfileId
Resource Attributes	Dublin-core based Metadata describing the resource that the profile describes, extended to include elements such as resource location and resource class.	Creator, Resource Class, Title

Profile Elements	Domain-specific metadata describing the resource which the profile describes.	Target Name

Much like Product Servers, each Profile Server contains zero or more *Query Handlers* which accept an OODT query, and then translate it into a query against the profiles which the Profile Server contains. Profile Servers can have backend data stores which store profiles, file systems which store profiles, or they can reference URL's which store file-based profiles, and thus there exists a need for a query handler mechanism to retrieve and query the back end profile store to retrieve profiles which satisfy the query criteria. Profile Query Handlers perform this task. Figure 4 shows the internal structure of the Profile Server Component.

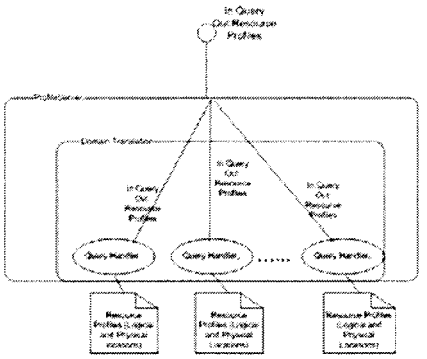

Figure 4. Internal structure of the Profile Server Component.

2.1.3 Query Servers

Query Servers are responsible for "putting it all together". Essentially a Query Server is bootstrapped with an initial set of "root" Profile Servers, much like Berkeley's DNS (BIND 2004). Upon receiving an OODT-style query, the Query Server proceeds to ask each one of the Profile Servers it knows about to retrieve the profiles which match resources which satisfy the query. This polling of Profile Servers is performed by *Profile Client* components which are present in an initial set of *Querier* threads. Essentially, the Profile Client is given a location for the Profile Server, and is able to query the profile server, and retrieve the list of profiles (if any) that match up resources that could satisfy the particular query. Each list of profile(s) returned by the Profile Client is passed to the Querier thread for examination of the Resource Class attribute in each profile. There are three cases which occur during this examination:

- **The Resource Class points to another Profile Server:** In this case, the Querier creates another instance of a Querier thread, seeded with the returned Profile Server resource location attribute as its root Profile Server.
- **The Resource Class points to a Product Server:** In this case, the Querier creates a Product Client component instance, which it seeds with the resource location attribute of the returned profile. The Product Client component retrieves the data products from the Product Server, and then the Product Client returns the data products back to the Querier thread. The Querier thread then adds its data products to the collected data products list in the Query Server, and ends its thread lifespan.
- **The Resource Class points to an individual data resource:** In this case, the Querier retrieves the individual data product and adds it to the collect data products list in the Query Server and ends its thread lifespan.

Once all the Querier threads have ended their respective lifespan, the Query Server is done collecting data products in its collected list. The collected data product list is then returned to the original user query. Figure 5 depicts the internal structure of the Query Server.

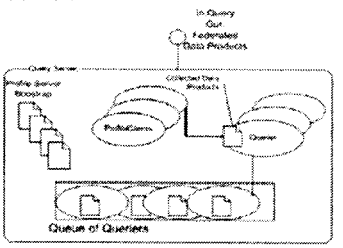

Figure 5. Internal structure of the Query Server Component.

2.2 Connectors

OODT supports one type of connector, a Messaging Layer, which allows OODT components to communicate with each other. The main form of communication between OODT components is the *Query Object*. A Query Object contains the Query q (mentioned in Section 2.1.1), and the query result, essentially the collected list of data products (mentioned in Section 2.1.3). The Query Object also contains a set of associated metadata regarding the query, such as *Maximum Accepted Results*, *Accepted Mime Types* (of Data Products within the collected data products list), and *Query Description*.

2.2.1 Messaging Layer

The OODT Messaging Layer's responsibility is to route Query Objects between Query Server, Product Server, and Profile Server Objects, and return the Query object to the correct User Client (typically the client which contacted the Query Server). The messaging layer essentially creates a virtual data bus which allows many Query Objects to be transferred between OODT components in a variety of different Messaging Protocols discussed below:

- **Unicast**: The Query Object is sent from an originating OODT component to exactly one other OODT component.
- **Multicast**: The Query Object is sent from an originating OODT component to a group of n other OODT components, defined by some common attribute, e.g. a type of Component URN (RFC3613 2003).
- **Broadcast**: The Query Object is sent from an originating OODT component to all other OODT components that the originator knows of.

Since Query Objects contain both the originating user query, as well as the collected data products which satisfy the originating query itself, the Query Object contains all the necessary information to generate important user satisfaction metrics such as information quality and query time.

2.3 Configurations

In this section, we discuss the possible configurations of OODT components in a real world situation. Since we discuss implementation level deployments at length throughout the rest of this paper, this discussion is purposefully brief and is not intended to be all encompassing, or to cover the entire spectrum of configurations that are possible with this architectural style. To motivate and conclude our discussion of this architectural style, we discuss a possible configuration of OODT in the context of linking together 4 different types of science data resources.

In the OODT architectural style, Product Servers are attached to an underlying Data System or Data Resource (such as a relational database, or an HTML web site, or a network file system). This relationship is shown in Figure 6, an augmented view of Figure 1, with Product Servers added to the diagram.

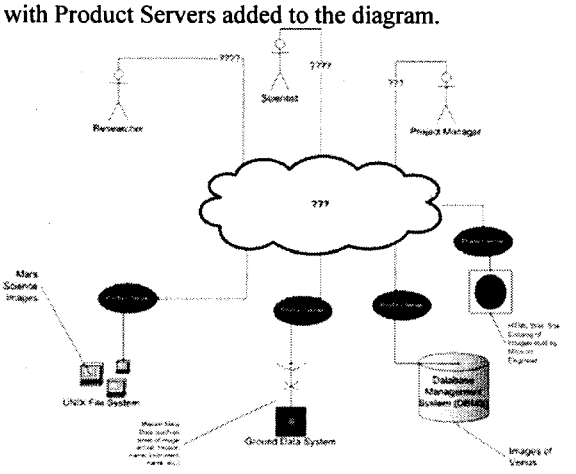

Figure 6. Augmented view of DIS scenario, with Product Servers depicted as red, shaded figures.

The Product Servers alone; however, do not fully implement the system. The system needs a method for discovery of Product Servers, and for federation of Products returned from each data source in the underlying system. In Figure 7, we augment the view of the DIS scenario from Figure 6 further, and begin to address this issue.

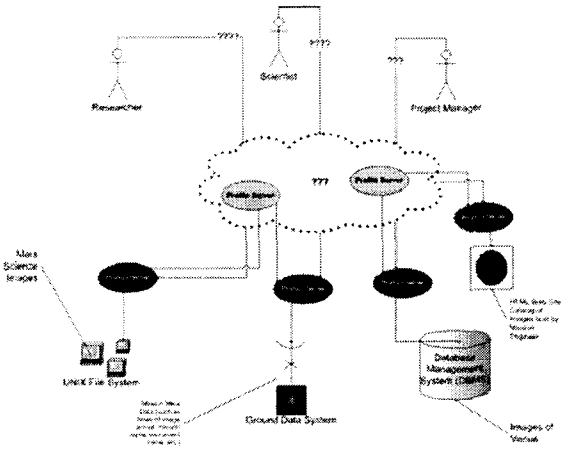

Figure 7. Augmented view of DIS scenario, with Profile Servers depicted as blue, lightly shaded figures.

In this scenario, each Profile Server *profiles* two Product Servers, along with information regarding what types of Data Products are stored in each Product Server (in order to satisfy the profile queries). The two profile servers lay infrastructure in terms of having the ability to locate and discover data resources which satisfy query parameters, but one more piece of infrastructure, the Query Server, will serve to tie the system components together to support the integrated system. This final piece of the configuration is shown in Figure 8.

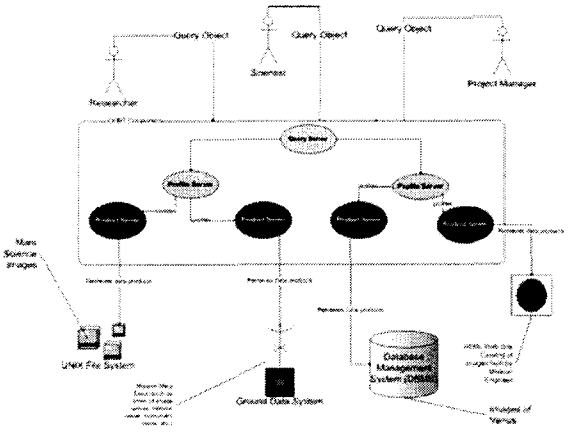

Figure 8. Augmented view of DIS scenario, with Query Servers depicted as yellow figures.

Figure 8 depicts a full configuration of OODT components. A *full configuration* is a configuration such that it can support querying from multiple, heterogeneous resources, as if they were one integrated virtual data source. The OODT style does not mandate that full configurations are the only existing configurations from style. Each system component (Query Server, Product Server, and Profile Server) are allowed to exist in a vacuum, but each alone will not necessarily realize the full capabilities of the style. Profile Servers and Product Servers alone may be queried in vacuum's as long as their location is known to the respective client component (Product or Profile Client).

The validation of this style is discussed with respect to its implementation middleware, and the middleware's respective successful large-scale deployments in both planetary science, as well as bio-medicine. We realize that there exist many further areas of research within the OODT style, as well as within architectural styles for data-intensive systems as a whole. OODT open areas of research with respect to style are further discussed in Section 6, as well as Section 7.

3. Architecture-based Middleware

The core OODT middleware has been implemented in Java, using Sun's Java Development Kit version 1.4. Programmers extend OODT core component classes such as *Profile Servers, Product Servers*, and *Query Servers* if additional functionality is needed, otherwise, the core classes are ready to use out of the box.

For the implementation of the Product Server component, programmers are required to write *QueryHandler* classes which implement the `jpl.oodt.product.[layer].QueryHandler` interface. *Layer* represents the type of messaging layer connector the programmer wants to use. Currently, CORBA and RMI exist, and we are working on a SOAP-based messaging layer at the time of writing this paper. The interface method to a query handler for a Product Server is the *query* method. The query method takes as input a Query Object (described in Section 2.2) which we can represent and serialize in XML (in the case where the distributed object middleware does not support more than the primitive data type parameters, such as in CORBA). The Query Object contains the query q (described in Section 2.1.1), and the associated query *metadata* (described in Section 2.2), and returns the Query Object back to the user, this time including any results which satisfied the query to the underlying data resource. The query interface method becomes the abstract translation function $t(q)$ discussed in section 2.1.1. We present below an XML DTD definition of our query object structure.

```
<!ELEMENT query
      (queryAttributes,
          queryResultModeId,
          queryPropogationType,
       queryPropogationLevels,
          queryMimeAccept*,
          queryMaxResults,
          queryResults,
          queryKWQString,
       queryStatistics?,
          querySelectSet,
          queryFromSet,
       queryWhereSet,
       queryResultSet)>

      <!ELEMENT queryAttributes
        (queryId,   queryTitle*,   queryDesc*,
queryType*,
         queryStatusId*, querySecurityType*,
         queryParentId*, queryChildId*,
         queryRevisionNote*,
         queryDataDictId*)>

      <!ELEMENT queryStatistics (statistic*)>

      <!ELEMENT querySelectSet
       (queryElement*)>

      <!ELEMENT queryFromSet
       (queryElement*)>

      <!ELEMENT queryWhereSet
       (queryElement*)>

        <!ELEMENT queryElement
          (tokenRole*, tokenValue*)>

      <!ELEMENT statistic (url, time)>

      <!ELEMENT queryResultSet
       (resultElement*)>

        <!ELEMENT resultElement
          (resultId*, resultMimeType*,
           profId*,                identifier*,
resultHeader,
           resultValue*)>

      <!ELEMENT                    resultHeader
(headerElement*)>
      <!ELEMENT    headerElement    (elemName,
elemType?, elemUnit?)>
```

The Product Server itself supports the exact same query method in its interface. When a query comes into the product server (via CORBA or RMI), the Product Server creates worker threads which each are given a particular Query Handler object reference in the List of Product Server Query Handlers, as well as a reference to the Product Server query method's xml query object. In parallel, each worker thread invokes its QueryHandler's query method, and passes it the reference to the Product Server's xml query object, so that each query handler upon return will aggregate the list of data return in the results section of the query object. To illustrate the ease with which query handlers can be written in the OODT middleware, we present a snippet of code below for a bare-bones query handler, which simply takes a query q, ignores the query, and adds one result to the result list of the query object.

```
import jpl.oodt.product.QueryHandler;
import java.io.*;

public class QueryHdlr implements QueryHandler
{

   public XMLQuery query(XMLQuery q) throws
ProductException{

     q.addResult(new Result());

   }
}
```

The Profile Server component implementation centers around the implementation of XML-based profiles, which profile data resources (and are described in Section 2.1.2 and Table 1). An XML DTD for the profile structure is given below:

```
<!ELEMENT profiles
(profile*)>
<!ELEMENT profile
(profAttributes,
resAttributes,
profElement*)>
<!ELEMENT profAttributes
(profId, profVersion?, profType,
profStatusId, profSecurityType?,
profParentId?, profChildId*,
profRegAuthority?, profRevisionNote*,
profDataDictId?)>
<!ELEMENT resAttributes
(Identifier, Title?, Format*, Description?,
Creator*, Subject*,
Publisher*, Contributor*, Date*, Type*,
Source*,
Language*, Relation*, Coverage*, Rights*,
resContext+, resAggregation?, resClass,
resLocation*)>
<!ELEMENT profElement
(elemId?, elemName, elemDesc?, elemType?,
elemUnit?,
elemEnumFlag, (elemValue* | (elemMinValue,
elemMaxValue)),
elemSynonym*,
elemObligation?, elemMaxOccurrence?,
elemComment?)>
```

The Profile Server functionality is implemented by writing Profile Server query handlers, which return back resource profiles in the above XML format. The Profile Server itself provides two interface methods, (1) A query method which accepts an xml based query object and returns resources profiles which implement the Common Data Elements used in query object's query string and (2) A getProfile method which accepts a profile Id, and returns the exact profile which matches the given id. The query method given in 1 essentially returns profiles by polling (in parallel using

the worker thread method discussed for the Product Server implementation) its respective Profile Query Handlers. Each QueryHandler implements the same 2 methods as its parent Profile Server, which allows each QueryHandler to return profiles asynchronously back to the parent.

The core Profile and Product Server class implementations are relatively small, averaging *56* SLOC each. This is due in part to the fact that programmer's typically write OODT Product and Profile Query Handlers, instead of extending the Profile and Product Server base components themselves each time. Since their respective query interaction mechanisms are relatively the same (we say *relatively* because one returns data product results, and one returns resource profiles, but both accept the same query object structure), then we felt that the domain translation mechanism should be the onus of the Query Handlers.

Upon creating Query Handlers that return both data products, and profiles, the method for linking the Query Handlers to the Product and Profile Servers respectively can be performed via XML configuration files, or at system-runtime. The XML configuration file essentially sets each Product and Profile server specified to have the associated Query Handlers, and instantiates each base component. After instantiation, the system determines what type of messaging connector is used (either RMI or CORBA), and then registers the distributed Profile or Product Server component with the appropriate registry (in RMI, Sun's RMI Registry can be used, in CORBA, the CORBA Name Service is used). Query Server components can then be instantiated and given the list of Profile Servers which exist in the system. Each Profile, Product, and Query Server is identified in the form of URNs, which uniquely identify the distributed component in the respective registry type. Our URN's are in the form of `urn:oodt:[layer]:[component-name]`. The respective layer registry enforces the requirement that the combination of [layer]: [component-name] must be unique when components register at run-time.

The implementation of the Query Server component typically comes last, which is directly tied to its architectural style notion of "bringing it all together". The Query Server component is seeded (either at run-time via configuration files or compile time) with a set of root Profile Server components (identified by URNs described in the next paragraph). The Query Server component supports one interface method, a query method. The query method accepts a query object as a parameter, and returns back the same query object this time containing the federated results that it was able to collect from the OODT connected system. To perform this result collection, the Query Component instantiates a base set Querier threads (mentioned in Section 2.1.3) which are each given a root Profile Server in the Query Server component's initial list. Each Querier thread proceeds in parallel to query its profile server and retrieve resource profiles by which to discover locations of data products which satisfy the original query. The Querier threads use the algorithm described in Section 2.1.3 to examine each resource profile's Resource Class attribute to determine how to go about reaching an eventual data product. Once all the original and any additional Querier threads which were created are finished collecting products, the method returns the query object with all the collected results.

4. Experience and Evaluation

In this section, we discuss deployments of the OODT middleware in two diverse domains, *Planetary Science*, with NASA's Planetary Data System (Hughes & McMahon 1998), and *Cancer Research*, with NIH's Early Detection Research Network (EDRN).

4.1 Planetary Data System

The Planetary Data System (PDS) manages and archives planetary science data for NASA's Office of Space Science. It has been in existence since the late 1980's and to this point has collected approximately six terabytes of data, which pre-OODT was distributed and archived for scientists on CD and DVD media.

The PDS is divided up into seven disciplined "nodes", which are geographically distributed across the country. Each node is responsible for managing and distributing its own particular planetary science data which is cumbersome due to their geographically diverse locations, and different methodologies (discussed below) for storing and accessing their data. Figure 10 illustrates the geographic diversity of the PDS.

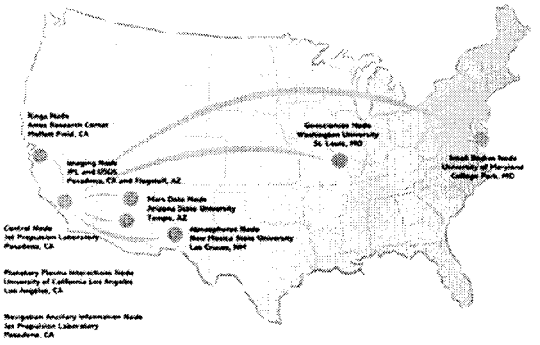

Figure 10. The Planetary Data System OODT Deployment.

The OODT middleware was used in 2002 to deploy an infrastructure for the distribution of PDS data from all seven nodes via the internet. OODT Product Server components wrapped data resources at each PDS Node (a total of 7 Product Servers) and a master set of Profile Servers were used to profile the data resources exposed by the OODT Product Server components. To tie everything together, a web portal search page was constructed to allow a user to pose a federated query across the entire PDS, and receive back all the PDS data and metadata which satisfies the query. The web portal search page poses its queries to an OODT Query Server, seeded with the master set of Profile Servers mentioned above.

To measure the amount of data transferred by the PDS OODT Deployment, we have employed *RecorderThread* components at each PDS Product Server which monitor the amount of data traffic for each query transaction per Product Server in the system. Since 11/15/2002, the PDS Product Server components have transferred 25GB of PDS Data Products to Scientists across the country (note that the 25GB so far are not including the full distributions of Mars Global Surveyor, Mars Odyssey, and MRO) Table 2 summarizes the metrics that we have collected to date regarding the PDS OODT Deployment.

Table 2. PDS Deployment total bytes and Products served since 11/15/2002.

Table 3 helps to illustrate the benefit of OODT in the PDS. Before OODT, PDS data was distributed via CD and DVD media, and the NASA PDS nodes had to mail out CDs and DVDs to scientists which contained the planetary science data from the missions which they desired to study. Since OODT, the data can now be distributed, and accessed via the web. In Table 2, we show 3 NASA/JPL missions, Mars Global Surveyor (MGS), Mars Odyssey (Odyssey), and Mars Reconnaissance Orbiter (MRO) and the amount of science data (in terabytes and gigabytes) that each mission was set to produce. The last column, *Cost* shows the estimated cost for distributing those PDS products on DVD media, and mailing out the PDS data to the scientists. In the case of MRO, OODT has saved NASA $186 million dollars in mailing and DVD costs, clearly an enormous savings.

Mission	Size TB	Size GB	Cost
MGS	0.5	600	$500K
Odyssey	4.0	4,000	$3.3M
MRO	224.0	224,000	$186M

Table 3. PDS mission data: expected volumes, and relative cost pre-OODT.

4.2 Early Detection Research Network

The National Cancer Institute's Early Detection Research Network (Kincaid et al 2002) is a network of over 30 cancer research centers participating in research geared towards the early detection of cancer. Particularly of interest to EDRN is cancer biomarker data (Kincaid et al 2002). Similar to the PDS, each EDRN site is geographically distributed across the United States, and each site manages their own data which does not interoperate and commingle with the other site's data. The ability to correlate this information is critical to cancer research (Kincaid et al 2002) in that it has been shown that as study volumes increase, so does the rate of scientific discovery. Also, in terms of validating and testing biomedical data, it is important to compare and contrast similar data at different EDRN sites.

The OODT middleware is currently supporting EDRN by providing the middleware infrastructure to tie together the distributed cancer research data located across EDRN's sites. Product Servers at 9 of the current EDRN locations wrap site specific data sources, and expose the cancer data to the overall system. A Profile Server network, located at Fred Hutchinson Cancer Research Center in Seattle provides the means for discovery of the existing Product Servers in the network. A portal web page, the ERNE (EDRN Resource Network Exchange) (ERNE 2002) provides web-based access to the OODT-based infrastructure. Figure 11 depicts the

EDRN architecture, and Figure 12 depicts the 9 sites tied together by the OODT middleware.

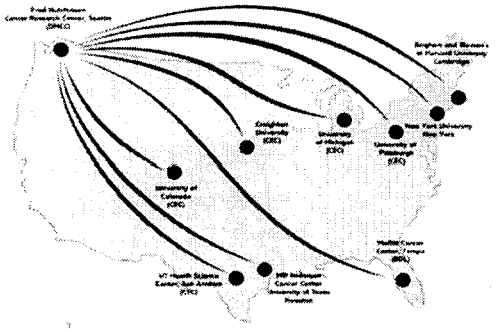

Figure 11. Nine sites connected by the OODT middleware in the geographically distributed Early Detection Resource Network (EDRN).

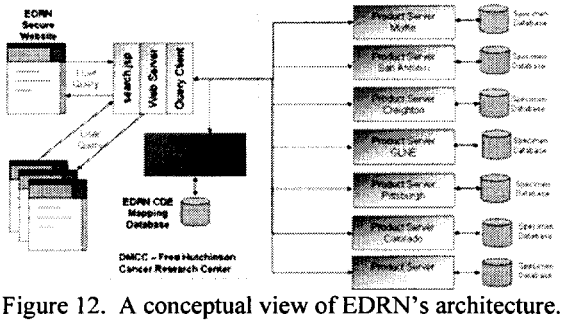

Figure 12. A conceptual view of EDRN's architecture.

5. Related Work

Work by (Kolp & Mylopolous 2001) models information systems using business-organizational structure as an architectural style, and argues that Multi-agent Systems (MAS) should be considered as a construct for information system architectural styles. They focus on business information systems; however, which are clearly a subset of data-intensive systems, and not broadly applicable. Also, (Kolp & Mylopolous 2001) focus only on requirements engineering for information systems, and conceptual architecture, whereas our approach focuses on architecture styles, and architectural-based middleware support, and system evolution.

(Kerschberg et al 1996) and (Gomaa & Menasce & Kerschberg 1996) present a novel information and data architecture for describing large-scale data-intensive information systems, specifically applied to NASA's EOSDIS science domain. Their work is most related to ours, but falls short in terms of a new architectural style as their work focuses only on the federated client-server (Fielding 2000) style for data-intensive systems. Also they present no fine-grained mapping of conceptual architecture to deployment architecture, or middleware-based solutions for implementing the data-intensive system. Of note, the EOSDIS system also suffered from many design flaws, which are discussed in (Leath 1998).

(Moore et al 1999) define Data-intensive systems as systems which are IO-bound. (Moore et al 1999) go on to describe the SDSC[1] Storage Request Broker (SRB) and how it can be used to abstract domain-specific data sources using a layered architectural style (Shaw and Garlan 1996). (Moore et al 1999) also presents a table[2] defining a basic set of application requirements for data-handling environments. There exists no mapping of these requirements; however to architectural components, nor to implementation-level constructs. Our approach aims to perform such work through the software engineering lifecycle for data-intensive systems. Also, as shown below in the related work for Grids, layered service architectures provide no insight as to the topologies of software architectural building blocks, such as components and connectors. Layered Service Architectures also typically present little guidance as to the legal configurations or behaviors of components, as laid out in deployment architectures, which is another aspect that we intend to research.

Recent work in the Grid Community (Globus 2004) has characterized a class of distributed data interoperable systems as Data Grids (Chrevenak 2000). Data Grids are discussed architecturally in terms of a layered services architecture (Shaw and Garlan 1996), and web-serviced based (Webservices.org 2004) middleware implementation. This approach is similar to ours in the sense that the architecture is developed first, and then middleware is instantiated to implement the architectural constructs. There is no mapping; however, of system requirements, to architectural components, to implementation level middleware in the Data Grid community, as (Casanova 2002) notes where he states that initial efforts in Data Grids (and the Grid community as a whole) were focused on "getting it to work" rather than system scalability, evolution, or design.

(Singh et. al 2003) defines a Metadata Catalog Service component, but does not describe its relationship in full to the layered services architecture that is implemented by the Globus toolkit (Globus 2004), nor to its deployment and interactions with other Grid components. (Singh et al. 2003) focuses on

---

[1] San Diego Supercomputing Center (http://www.sdsc.edu)

[2] Table 5.2 in (Moore et al 1999)

the component's scalability, which is indicative of the Grid community's tendency to focus on super-computing challenges, as opposed to formal software engineering methodology and software architectural research. Our work focuses on uniting formal software engineering principles with the design of data-intensive systems such as Data Grids.

6. Open Issues

There remain several open issues with OODT that we will address briefly in this section. First and foremost, architecting, and deploying OODT software architectures, and subsequent middleware is very *programmer-intensive*. By "programmer-intensive" we mean to imply that a programmer is required to be "in the loop" in order to successfully deploy and architect these systems. This is due to the fact that the programmer is responsible for translating OODT architectural constructs into extensions of our existing middleware framework (discussed in Section 4). Typically, a programmer will be involved in the early phases of the software process, helping to gather requirements, and translate requirements into some tailoring, configuration, deployment of existing OODT code.

One way of addressing problems like these has typically been to provide architectural design tools, such as UC Irvine's xADL (Dasofhy & Van der Hoek & Taylor 2002), for software designers to create architectural diagrams, and then have a way of mapping those architectural diagrams to software implementation and code deployment. There is existing work in this area (Medvidovic et al. 2003), and we aim to research and construct tool support to model and deploy architectures and software systems in the OODT-style.

Another open-issue in OODT is the issue of middleware deployment metrics, such as scalability of hundreds of components deployed across a set of different hosts. We are currently working on generating metrics for research in this area, and initial results look promising.

The OODT middleware also assumes a reliable network is present in order for the Product, Profile, and Query Servers to communicate across. We currently have no support for issues such as disconnected operation (Mikic-Rakic & Medvidovic 2003), and off-loading of data to support unreliable hosts. This type of fault-tolerance is crucial in data-intensive systems which may be deployed in unreliable environments.

7. Future Research

Our future research centers on a software-architecture focused approach to design, implementation, and evolution of data-intensive systems. We also are studying software architecture for generally describing the nature of integrating heterogeneous data sources. An end-to-end, and re-useable methodology for this problem, to our knowledge, has not been developed, and existing research has not taken this type of focused approach to creating a solution to this problem. Potential future work includes:

1. What types of architectural styles (Fielding 2000) exist for data-intensive systems, and how can they be re-used, learned from, and put to use in designing, implementing, and evolving DIS?
2. Does a set of standard components, connectors (Mehta & Medvidovic & Phadke 2000), and configurations exist which can describe data-intensive systems?
3. Can product-line architectures (Clements & Northrop 2001) be developed for data-intensive systems?
4. How well does UML describe data-intensive systems?
5. Can an ADL (Architecture Description Language) (Medvidovic & Taylor 200) be developed to describe data-intensive systems?

At JPL and USC, we are researching these questions and look to provide guidance, research experience and working prototypes for investigating these important issues.

8. Conclusions

We have presented OODT, an architectural style and middleware implementation for data-intensive systems. We feel that data-intensive systems have been a neglected area of research in the software engineering and software architecture communities, and we desire to apply formal software architectural methodologies to the design, implementation, and evolution of data-intensive systems. The OODT style and middleware was developed at NASA's Jet Propulsion Laboratory and has been supporting NASA's Planetary Data System in the Planetary Science Domain. The OODT middleware also supports Cancer Research, and is currently deployed at

the National Institute of Health's National Cancer Institute, supporting the EDRN (Early Detection Resource Network) task. From our experience, the area of software architecture for data-intensive systems appears to be a fruitful area of research for the years to come.

9. Acknowledgments

This work was performed at the Jet Propulsion Laboratory, managed by the California Institute of Technology, under contract with the National Aeronautics and Space Administration (NASA). Copyright © 2004. This work was also supported by <insert Neno grant here>.

10. References

Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead, Jr., Jason E. Robbins, Kari A. Nies, Peyman Oreizy, and Deborah L. Dubrow. "A Component- and Message-Based Architectural Style for GUI Software." *IEEE Transactions on Software Engineering*, vol. 22, no. 6, pages 390-406 (June 1996).

R. W. Moore et al. "Data-Intensive Computing". *In The Grid: Blueprint for a New Computing Infrastructure.* Morgan-Kaufman Publishers, 1999.

D Crichton, J.S. Hughes, S. Kelly. "A Science Data System Architecture for Information Retrieval". *In Clustering and Information Retrieval.* Kluwer Academic Publishers. June 2003.

H.M. Sneed. "The Rationale for Software Wrapping". *In Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, October 01-03, 1997, p. 303.

DCMI, "Dublin Core Metadata Element Set, Version 1.1: Reference Description," Dublin Core Metadata Initiative, 1999.

ISO/IEC, "Framework for the Specification and Standardization of Data Elements 11179-1," Specification and Standardization of Data Elements 11179, International Organization for Standardization, Geneva, 1999.

J. Steven Hughes, Susan K. McMahon. "The Planetary Data System. A Case Study in the Development and Management of Meta-Data for a Scientific Digital Library". *In Proceedings of the European Conference on Digital Libraries (ECDL)*, 1998. pp. 335-350

Berkeley Internet Name Domain (BIND). http://www.isc.org/bind.html, 2004.

Extensible Markup Language (XML) 1.0 (Second Edition) W3C Recommendation 6 October 2000, http://www.w3.org/TR/2000/REC-xml-20001006

RFC 3613. "Definition of a Uniform Resource Name (URN) namescape", 2003. http://www.faqs.org/rfcs/rfc3613.html

Marija Mikic-Rakic and Nenad Medvidovic. "Toward a Framework for Classifying Disconnected Operation Techniques". *In Proceedings of the Second International Workshop on Software Architectures for Dependable Systems (WADS'03)*, Portland, Oregon, May 2003.

E.M. Dashofy, A. Van der Hoek and R.N. Taylor. "An Infrastructure for the Rapid Development of XML-based Architecture Description Languages". *In Proceedings of the 24th International Conference on Software Engineering (ICSE), p.266*-267, 2002.

Nenad Medvidovic, Marija Mikic-Rakic, Nikunj Mehta, and Sam Malek. "Software Architectural Support for Handheld Computing". Cover feature in *IEEE Computer* , September 2003.

R. T. Fielding. *Architectural Styles and the design of Network-based Software Architectures*, Ph.D. Dissertation, University of California, Irvine, 2000.

Nenad Medvidovic and R. N. Taylor. "A Classification and Comparison Framework for Architecture Description Languages". *IEEE Transactions on Software Engineering*, Vol 26. No 1. January, 2000, pp. 70-93.

Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. "Towards a Taxonomy of Software Connectors". *In Proceedings of the International Conference on Software Engineering (ICSE)*, Limerick, Ireland, June 2000.

P. Clements & L. Northrop. *Software Product Lines: Practices and Patterns.* Boston, MA: Addison-Wesley, 2001.

Webservices.org. http://www.webservices.org, January, 2004.

Henri Casanova, "Distributed Computing Research Issues in Grid Computing", ACM SIGAct News, Vol. 33, No. 3, 2002, pp. 50 -- 70. http://citeseer.nj.nec.com/casanova02distributed.html

L. Kerschberg et al. Data and Information Architectures for Large-scale Data Intensive Information Systems. *In Proceedings Eighth International Conference on Statistical and Scientific Database Management*, Stockholm, Sweden, June 18-20, 1996

Audrey T. Leath. NASA's Earth Science Programs come under Scrutiny. *The American Institute of Physics Bulletin of*