XML Metadata Services

Mehmet S. Aktas[1,2], Sangyoon Oh [1,2], Geoffrey C. Fox[1,2,3], Marlon E. Pierce[1]

[1]Community Grids Laboratory
[2]Department of Computer Science
[3] Department of Physics
Indiana University
{maktas, ohsangy, gcf, mpierce}@cs.indiana.edu

Abstract: As the Service Oriented Architecture (SOA) principles have gained importance, an emerging need has appeared for methodologies to locate desired services that provide access to their capability descriptions. These services must typically be assembled into short-term service collections that, together with code execution services, are combined into a meta-application to perform a particular task. To address metadata requirements of these problems, we introduce a hybrid Information Service to manage both stateless and stateful (transient) metadata. We leverage the two widely used web service standards: Universal Description, Discovery, and Integration (UDDI) and Web Services Context (WS-Context) in our design. We describe our approach and experiences when designing "semantics". We report results from a prototype of the system that is applied to mobile environment for optimizing Web Service communications.

1. Introduction

As the Service Oriented Architecture (SOA) principles have gained importance, an emerging need has appeared for methodologies to locate desired services that provide access to their capability descriptions. As these services interact with each other within a workflow session to produce a common functionality, another emerging need has also appeared for storing, querying, and sharing the resulting metadata needed to describe session state information.

Zhuge identifies the two mainstream research focuses in next-generation Web in [1]. The first research theme investigates how to overcome the existing Web's limitations such as difficulties in supporting intelligent services. Some example areas of investigation of this approach are Semantic Web and Web Services. The second research theme focuses on the Grid as an alternative application platform. The Grid offers a model for solving computational science problems by utilizing the idle resources of large numbers of distributed computers. Zhuge also mentions the Semantic Grid research, as an extension to the Grid, evolved as result of integration of the two aforementioned mainstream research themes.

As the SOA-oriented architectures gained popularity in both the traditional and Semantic Grid, metadata management problems of Grid applications form an important area of investigation. For an example, Geographical Information Systems (GIS) provide very useful problems in supporting "virtual organizations" and their associated information systems. These systems are comprised of various archival data services (Web Feature Services), data sources (Web-enabled sensors), and map generating services. All of these services are metadata-rich, as each of them must describe their capabilities (What sorts of features do they provide? What geographic bounding boxes do they support?). Organizations like the Open Geospatial

Consortium define these metadata standards.

These services must typically be assembled into short-term, stateful service collections that, together with code execution services and filter services (for data transformations), are combined into a composite application (e.g. a workflow).

To address metadata requirements of these problems, we introduce XML Metadata Services to manage both stateless and stateful (transient) metadata. We use and extend the two Web Service standards: Universal Description, Discovery, and Integration (UDDI) [2] and Web Services Context (WS-Context) [3] in our design. We utilize existing UDDI Specifications and design an extension to UDDI Data Structure and UDDI XML API to be able to associate both prescriptive and descriptive metadata with service entries. We extend WS-Context specifications to provide search/access/storage interface to session metadata.

In this paper, we describe the "semantics" of the proposed approach and give an overview of implementation details. In addition, we also discuss a motivating application scenario and the way that the hybrid system is being used. We report results from a prototype that has been applied to mobile environment for optimizing Web Service communications.

2. Background

There have been some approaches introduced to provide better retrieval mechanism by extending existing UDDI Specifications. UDDI-M [4] and UDDIe [5] projects introduce the idea of associating metadata and lifetime with UDDI Registry service descriptions where retrieval relies on the matches of attribute name-value pairs between service description and service requests. In our design, we too extend UDDI's Information Model, by providing an extension where we associate metadata with service descriptions similar to existing solutions where we use name-value pairs to describe characteristics of services. Apart from the existing methodologies, we provide both general and domain-specific query capabilities. An example for domain-specific query capability could be XPATH and RDQL queries on the auxiliary and domain-specific metadata files stored in the UDDI Registry.

The primary use of our approach is to support information in dynamically assembled workflow-style Grid applications where services are tied together in a dynamic workflow to solve a particular problem. There are varying specifications, such as WSRF [6], WS-Context, WS-Transfer [7], and WS-Metadata Exchange [8], that have been introduced to define stateful interactions among services. Among them, we have chosen the WS-Context specifications to create a metadata catalog system for storing transitory metadata needed to describe distributed session state information. Unlike the other specifications defining service communications, WS-Context models a session metadata repository as an external entity where more than two services can easily access/store highly dynamic, shared metadata.

3. Abstract Data Models

We have designed and built a novel architecture [9-10] for an hybrid Information Service supporting handling and discovery of not only quasi-static, stateless metadata, but also session related metadata. We based the information model and programming interface of our system on two widely used specifications: WS-Context and Universal Description, Discovery and Integration (UDDI).

We have identified following base elements of the semantics of proposed system: a) data semantics, b) semantics for publication and inquiry XML API, and c) semantics for security and access control XML API. These semantics have been designed under two constraints. First, both UDDI and WS-Context Specifications should be extended in such a way that client applications to these specifications can easily be integrated with the proposed system. Second, the semantics of the proposed system should be modular enough so that it can easily be operated with future releases of these specifications.

3.1. Extensions to UDDI Abstract Data Model

The extended version of UDDI information model consists of various additional entities to existing UDDI Specifications (Detailed design documents can be found at http://www.opengrids.org/extendeduddi). These entities are represented in XML. We describe extensions to UDDI information model as following: serviceAttributeEntity: A service attribute data structure describes metadata associated with service entities. Each “serviceAttribute” corresponds to a piece of metadata and it is simply expressed with (name, value) pairs. A “serviceAttribute” can be categorized based on custom classification schemes. A simple classification could be whether the “serviceAttribute” is prescriptive or descriptive. A service attribute may also correspond to a domain-specific metadata and could be directly related with functionality of the service. leaseEntity: A lease entity describes the lifetime associated with services or context. This entity indicates that the service or context will be considered alive and can be discovered by client applications until the lease expires.

3.2. WS-Context Abstract Data Model

Although WS-Context Specification presents XML API to standardize behavior and communication of the service, it does not define an information model. We introduce an information model comprised of various entities. Here, entities are represented in XML and stored by the service. The proposed information model composed of instances of the entities as following. sessionEntity: A session entity describes a period of time devoted to a specific activity, associated contexts, and services involved in the activity. A session can be considered as an information holder for the dynamically generated information. Each session is associated with its participant web services. Also, each session contains contexts which might be associated with either services or session or both. contextEntity: A context entity describes dynamically generated metadata that is associated either to a session or a service or both. leaseEntity: A lease entity describes a period of time during which a service or a context can be discoverable. A lease entity is associated to both session and context entities.

3.3. Extended UDDI and WS-Context Inquiry and Publication API Sets

We present extensions/modifications to existing WS-Context and UDDI APIs to standardize the additional capabilities of our implementation. We then integrate both extended UDDI and WS-Context API sets within a uniform programming interface: Hybrid Grid/Web Information Service. The API sets of the hybrid service can be grouped as following: ExtendedUDDI Inquiry, ExtendedUDDI Publication, WS-Context Inquiry, WS-Context Publication, WS-Context Security and Publisher XML APIs.

3.3.1. Extended UDDI Inquiry API

We introduced various APIs representing inquiries that can be used to retrieve data from the hybrid service as following: find_service: Used to extend the out-of-box UDDI find service functionality. The find_service API call locates specific services within the service. It takes additional input parameters such as serviceAttributeBag, contextBag and Lease to facilitate additional capabilities of the proposed system. find_serviceAttribute: Used to find aforementioned serviceAttribute elements. The find_serviceAttribute API call returns a list of serviceAttribute structure matching the conditions specified in the arguments. get_serviceAttributeDetail: Used to retrieve semi-static metadata associated to a unique identifier. The get_serviceAttributeDetail API call returns the serviceAttribute structure corresponding to each attributeKey values specified in the arguments.

3.3.2. Extended UDDI Publication API

We introduce various extensions to UDDI Publication API set to publish and update semi-static metadata associated with service. save_service: Used to extend the out-of-box UDDI save service functionality. The save_service API call adds/updates one or more web services into the service. Each service entity may contain one to many serviceAttribute and/or one to many contextEntity elements and may have a life time (lease). save_serviceAttribute: Used to register or update one or more semi-static metadata associated to a web service. delete_serviceAttribute: Used to delete existing serviceAttribute element from the service.

3.3.3. WS-Context Inquiry API

We introduce extensions to WS-Context Specification for both inquiry and publication functionalities. The extensions to WS-Context Inquiry API set are outlined as following: find_session: Used to find sessionEntity elements. The find_session API call returns a session list matching the conditions specified in the arguments. get_sessionDetail: Used to retrieve sessionEntity data structure corresponding to each of the session key values specified in the arguments. find_context: Used to find contextEntity elements. The find_context API call returns a context list matching the criteria specified in the arguments. get_contextDetail: Used to retrieve the context structure corresponding to the context key values specified.

3.3.4. WS-Context Publication API

We outline the extensions to WS-Context Specification Publication API set to publish and update dynamic metadata as following: save_session: Used to add/update one or more session entities into the service. Each session may contain one to many serviceAttribute, have a life time (lease) and be associated with service entries. delete_session: Used to delete one or more sessionEntity

structures. save_context: Used to add/update on or more context (dynamic metadata) entities into the service. delete_context: Used to delete one or more contextEntity structures.

3.4. Authentication Mechanism

In order to avoid unauthorized access to the system, we adopted semantics from existing UDDI Security XML API and implemented a simple authentication mechanism. In this scenario, each publication/inquiry request is required to include authentication information (authInfo XML element). Although this information may enable variety of authentication mechanisms such as X.509 certificates, for simplicity, we implemented a username/password based authentication scheme. A client can only access to the system if he/she is an authorized user by the system and his/her credentials match. If the client is authorized, he/she is granted with an authentication token. An authentication token needs to be passed in the argument lists of publication and inquiry functions, so that these operations can take place.

3.4.1. WS-Context Security API

We adopt the semantics from out-of-box UDDI Security API set in our design. The Security API includes following function calls. get_authToken: Used to request an authentication token as an "authInfo" (authentication information) element from the service. The autInfo element allows the system implement access control. To this end, both publication and inquiry API set includes authentication information in their input arguments. discard_authToken: Used to inform hybrid WSContext service that an authentication token is no longer required and should be considered as invalid.

3.5. Authorization Mechanism

When a context is published to the system, by default an owner-relationship is established between the publisher and the context. The owner of the context specify various permissions such as what access rights a) the owner, b) the members of the owner's group, and c) the rest of the users will have to the context. For each of these categories there exist read, write and read/write access rights. This basic security mechanism is also used in UNIX operating system. Upon receiving a request, the system checks access permission rights specified in a context, before granting inquiry/publication request to the context.

3.5.1. WS-Context Publisher API

We introduce various APIs to provide find/add/modify/delete on the publisher list, i.e., authorized users of the system. These APIs include the following function calls. find_publisher: Used to find publishers registered with the system matching the conditions specified in the arguments. save_publisher: Used to add or update information about a publisher. delete_publisher: Used to delete information about a publisher with a given publisherID from the metadata service. get_publisherDetail: Used to retrieve detailed information regarding one or more publishers with given publisherID(s).

Given these capabilities, one can simply populate the hybrid service with metadata as in the following scenario. Say, a user publishes a new service into the system. In this case, the user

constructs both “metadataBag” filled with “serviceAttributes” and “contextBag” filled with “contexts” where each context describes the sessions that this service will be participating. As both the “metadataBag” and “contextBag” is constructed, they can be attached to a new “service” element which can then be published with extended “save_service” functionality of the hybrid service. On receiving publishing service metadata request, the system applies following steps to process service metadata. First, the system separates the dynamic and static portions of the metadata. Then the system issues the static portion (“metadataBag”) of the query on the extended UDDI MySQL database, while it issues dynamic portion (“contextBag”) of the query on the WSContext MySQL database. Further design documentation on hybrid XML Metadata Service is available at http://www.opengrids.org.

4. An Application Usage Scenario: Handle Flexible Representation (HHFR) System

In order to present the applicability of our system, we briefly outline a metadata storage component (the Context-store) of an application use domain (mobile environment) in which the proposed hybrid system is used.

4.1. An Overview of the Service Oriented Architecture for HHFR System

A novel Web Service architecture, Handheld Flexible Representation (HHFR) is developed for optimizing Web Service performance in mobile computing which is physically constrained and requires an optimized messaging scheme to prevent performance degradations. Despite its important role in distributed computing, mobile computing hasn't reached its full potential because of the limited availability of high speed wireless connections (e.g. third generation cellular technology) as well as shortened the device’s use time when it is connected to a faster channel and do more computation. Thus, applying current Web Service communication models to mobile computing may result in unacceptable performance overheads caused by the encoding and decoding of verbose XML-based SOAP messages.

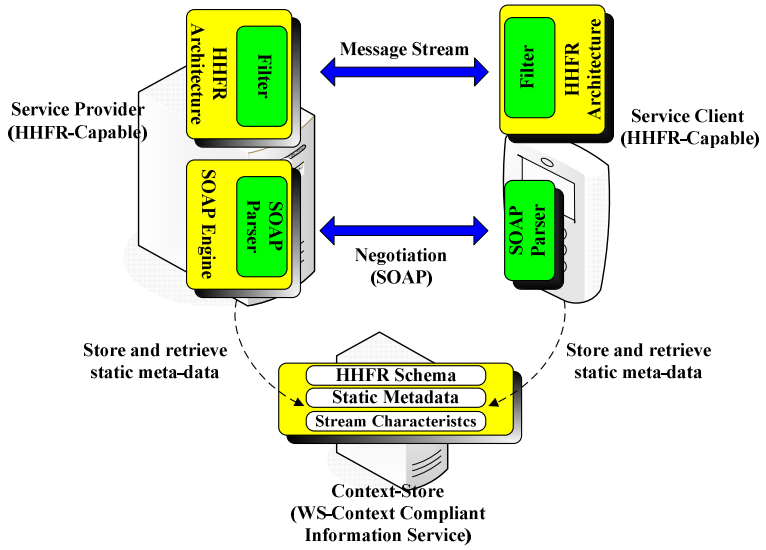

Figure 1 Overview of HHFR Architecture

HHFR let two end-points exchange information in a form of a message stream which is achieved on both semantic and protocol level by which mobile devices are able to overcome possible communication overheads caused by many factors: not only encoding/decoding overhead stated above, but also high latencies of mobile connection using HTTP. It provides a framework to negotiate characteristics of the message stream and representation between a mobile application and a corresponding service node.

Message streaming on semantic level is achieved by two methods. First, HHFR provides a scheme to separate the semantics of a message and its representation. Using Data Format Description Language (DFDL)-style data description language, HHFR describes the data format in Simple_DFDL language and the stub in the framework converts data from and to a preferred representation. The other method used by HHFR for building a message stream is storing unchanging/redundant message parts to hybrid information service (i.e. Context-store).

Let's consider a scenario where a user has a cell phone, which is running a videoconferencing application packaged as a "lightweight" Web Service. Such service could be a conferencing, streaming, or instant messaging service. To optimize service communication, the redundant/unchanging parts of the messages, exchanged between two services, must be stored on a third-party repository, i.e., Context-store.

The redundant/unchanging parts of a SOAP message are XML elements which are encoded in every SOAP message exchanged between two services. These XML elements can be considered as "context", i.e. metadata associated to a conversation. Here, hybrid XML Metadata Service is being used as the Context-store [11]. Each context is referred with a system defined URI where the uniqueness of the URI is ensured by the system. The corresponding URI replaces the redundant XML elements in the SOAP messages which in turn reduce the size of the message for faster message transfer. Upon receiving the SOAP message, the corresponding parties in service conversation interacts with the hybrid service to retrieve the context associated with the URIs listed in the SOAP message.

4.2. Usage of the Context-store

The Context-store archives the static context information from a SOAP negotiation message, such as unchanging or redundant SOAP headers, a Simple_DFDL document as a message representation, and the character of the stream. By archiving, the Context-store can serve as a meta-data repository for the participating nodes in the HHFR architecture (i.e. database semantic). The hybrid metadata service is essential to HHFR architecture since the service provides a method to store message parts and sharing them between messages in the stream makes them related each other and ultimately reduce the size and complexity of them. The Context-store in the architecture also guarantees a semantically persistent recovery from disconnections which is more common in intermittent mobile domain than conventional computing domain.

We integrate a hybrid metadata service with HHFR through a direct serialization of the SOAP request message and a parsing SOAP without Axis SOAP-Java binding. It is because Axis version for mobile environment is not developed yet. We use the kSOAP library for those processes.

4.3. Performance Improvements in HHFR with the Context-store

To show the effectiveness of using the hybrid metadata service (i.e. Context-store), we measured a round trip time of both the full SOAP message and the optimized message with Context-store usage. As stated, the size of the message can be reduced and the performance of the messaging can also be increased.

The experiment uses various sizes of SOAP example documents as message; the round trip times for a large message are collected using a WS-Security headers and a sample message with WS-Addressing headers is used for a medium size. In this experiment, the messages between mobile devices and services are exchanged over HHFR. The result show we save 83% of message size on average and 41% of transit time on average by using the Context-store. The results are shown in Table 1 and the configuration of the mobile devices and the service provider machine is given in Table 2 and Table 3 respectively.

Message Size	Full SOAP Message		Optimized Message	
	Ave.±error	Stdev	Ave.±error	Stdev
Medium: 513byte (sec)	2.76±0.034	0.187	1.75±0.040	0.217
Large: 2.61KB (sec)	5.20±0.158	0.867	2.81±0.098	0.538

Table 1 Summary of the Round Trip Time

Service Client: Treo 600	
Processor	ARM (144MHz)
RAM	32MB total, 24MB user available
Network Bandwidth	14.4Kbps (Sprint PCS Vision)
OS	Palm 5.2.1.H
Java Version	Java 2, Micro CLDC 1.1, MIDP 2.0

Table 2 Summary of Mobile Device Configuration

Service Provider: Grid Farm 8	
Processor	Intel® Xeon™ CPU (2.40GHz)
RAM	2GB total
Network Bandwidth	100Mbps
OS	GNU/Linux (kernel release 2.4.22)
Java Version	Java 2 platform, (1.5.0-06)

Table 3 Summary of Service Provider Machine Configuration

5. An Overview of the prototype implementation of the hybrid XML Metadata Service

We assume a range of applications which may be interested in integrated results from two different metadata spaces; UDDI and WS-Context. When combining the functionalities of these two technologies in one hybrid service, we may enable uniform query capabilities on context (service metadata) catalog. To this end, we have implemented a uniform programming interface, i.e. a hybrid information service combining both extended UDDI and WS-Context. (see Session

3 for detailed discussion on Information Model and XML API Sets of the hybrid service). Here, we give an overview of the system components, their functionalities and discuss how these components interact with each other.

Our implementation consists of various modules such as Query and Publishing, Expeditor, Access, Storage and Sequencer Modules. The Query and Publishing Module is responsible for performing operations issued by end-users. The Expeditor Module forms a generalized caching mechanism. One consults the expediter to find how to get (or set) information about a dataset in an optimal fashion. The Access and Storage modules are responsible for actual communication between the distributed Hybrid Services in order to form a distributed replica hosting environment. In particular, the Access module deals with client request distributions, while the Storage module deals with replication. Finally, the Sequencer Module is used to label each metadata which will be stored in the system.

When receiving a query, the Query and Publishing Module first processes the query. Then, it forwards the query to Expediter, where the Expeditor Module checks whether the requested data is in the cache. The Expeditor Module implements a generalized caching mechanism and forms a built-in memory. It utilizes the TupleSpaces paradigm [12] which is a space based programming providing mutual exclusive access that in turn enables data sharing between processes. For the purposes of this research, a tuple is termed as context and the tuplespaces as ContextSpaces. The Expeditor Module implementation is built on MicroSpaces libraries [13]. MicroSpaces is a free, open-source, and a light-weight implementation of TupleSpaces paradigm. The MicroSpaces codebase is expanded in the following ways in order to incorporate with our implementation. First, a context management scheme is implemented to manage storage and dynamic replication decisions for the contexts stored in the ContextSpace. This is succeeded by implementing a Java Thread which is responsible for a) checking the ContextSpace for updates with frequent time intervals, b) storing updated contexts into MySQL database and c) deciding on dynamic replica placements. Second, an Expeditor Handler library is implemented in order to query/publish data in local database. An Expeditor handler allows processes to do operations on the ContextSpace as the primary storage. As the system keeps all metadata keys in memory, if the Expeditor Module can not find the result, then the Query and Publishing Module will forward the query to Access Module, where the Access Module multicast a probe message to available services through a messaging infrastructure which is based on publish/subscribe paradigm. We use NaradaBrokering (NB) [14] software which is an open-source and distributed messaging infrastructure implementing publish/subscribe paradigm. This way the service communicates with the original data sources to satisfy the query under consideration. The query is responded by those services that host the matching context. At last, on receiving the results, the Query and Publishing Module returns the results to the querying client.

6. EVALUATION

We have performed experiments to investigate the performance and scalability of aforementioned hybrid service.

Machine Configurations	
Processor	Intel® Xeon™ CPU (2.40GHz)
RAM	2GB total
Network Bandwidth	100Mbps
OS	GNU/Linux (kernel release 2.4.22)
Java Version	Java 2 platform, (1.5.0-06)

Table 4 Summary of Linux Machine Configurations

We tested our code using various nodes of a cluster located at the Community Grids Laboratory of Indiana University. This cluster consists of eight Linux machines that have been setup for experimental usage. The cluster node configuration is given at Table 5. We wrote all our code in Java, using the Java 2 Standard Edition. In the experiments, we used Tomcat Apache Server with version 5.5.8 and Axis software with version 2 as a service deployment container.

Firstly we applied a performance experiment. The primary interest in doing this experiment is to understand the baseline performance of the implementation of hybrid Information Service. The performance evaluation of the service is done for inquiry and publication functions under normal conditions, i.e., when there is no additional traffic.

In this experiment, we particularly investigate performance of our caching methodology for WS-Context standard operations. We conduct following testing cases: a) A client sends queries to a dummy service. The dummy service receives a message and then sends it back to the client with no processing applied. b) A single client sends inquiry/publication requests to a hybrid Information Service where the system grants the request with memory access. c) A single client sends inquiry/publication requests to a hybrid Information Service where the system grants the request with database access.

This experiment studies the effect of various overheads that might affect the system performance. To do this, a dummy service, which is simply an echo service that returns the input parameter passed to it, is being used. This service helps measuring various overheads such as the network communication, client application initialization and container processing. By comparing and contrasting the results from the dummy service and the actual hybrid service, the actual time spent for pure server side processing can be observed. In this experiment, we use the same Web Service container engine (Apache Axis with version 2) for all testing cases.

In our investigation of system performance, we conducted the testing cases when there were 5000 metadata published in the system. At each testing case, the client sends 200 sequential requests for either publication or inquiry purposes. We record the average response time. This experiment was repeated five times. Figure 3 illustrates the system performance when the inquiry function was executed, while Figure 4 illustrates the same when the publication function was executed. The detailed statistics corresponding to these testing cases is listed in Table 7 and Table 8.

We also conduct an experiment where we investigate the best possible backup-interval period to provide fault-tolerance and high performance at the same time. Based on this experiment, we observe the trade-off in choosing the value for backup-time-interval. If the backup frequency is too high such as every 10 milliseconds, then the time required for a publication function is around 10.2 milliseconds. If the backup frequency is every 10 seconds or lower, we find that average execution time for publication operation stabilized to 7.46 milliseconds. Therefore, we choose the value for backup frequency as every 10 seconds in our experiments.

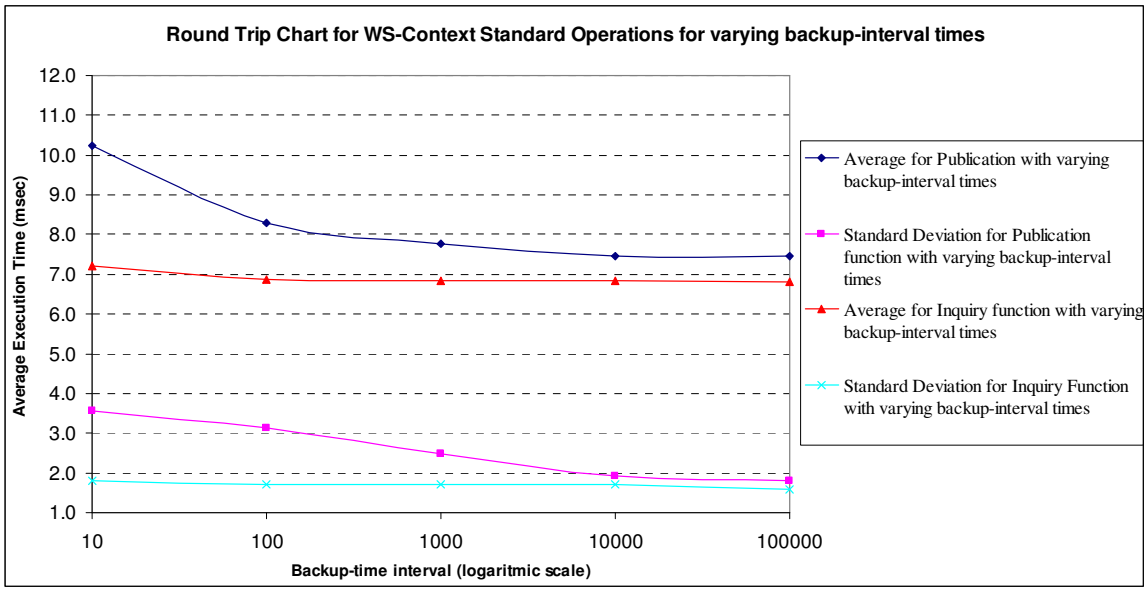

Figure 2 Test results for backup frequency investigation

	Publication Function		Inquiry Function	
Kbytes	mean	stdev	mean	stdev
0.01	10.24	3.57	7.20	1.80
0.1	8.29	3.13	6.86	1.71
1	7.76	2.48	6.85	1.70
10	7.46	1.94	6.85	1.71
100	7.46	1.82	6.81	1.60

Table 6 Statistics for the Figure 2

Figure 3 shows the performance results of inquiry function, while Figure 4 shows the performance results of publication function. The empirical results show that a) for inquiry function, we gain around 47% performance increase and b) for publication function, we gain around 30% performance increase by employing a cache mechanism in our design. This experimental study indicates that one can achieve noticeable performance improvements in metadata management for standard inquiry and publication operations by simply employing a memory built-in caching mechanism (the Expeditor Module), while preserving a certain fault-tolerance level as the contexts have to be backed up offline in at most N time unit. Based on our

investigation on backup frequency, we choose the value of N to be 10 seconds.

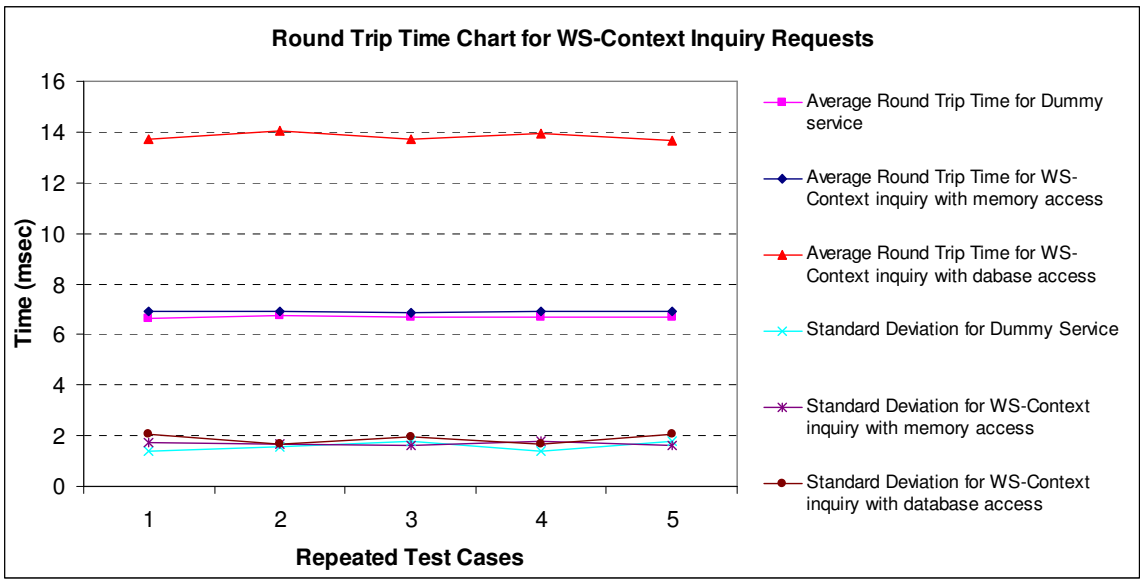

Figure 3 Round Trip Time Chart for Inquiry Requests

Statistics for the first test set from different publication request testing cases		
	mean	stdev
Test-1	6.64	1.40
Test-2	6.92	1.70
Test-3	13.73	2.08

Table 7 Statistics for the first test set from each testing cases conducted to test inquiry operation performance. (Test-1: Dummy service testing case, Test-2: WS-Context inquiry with memory access testing case, Test-3: WS-Context inquiry with database access testing case). The time units are in milliseconds.

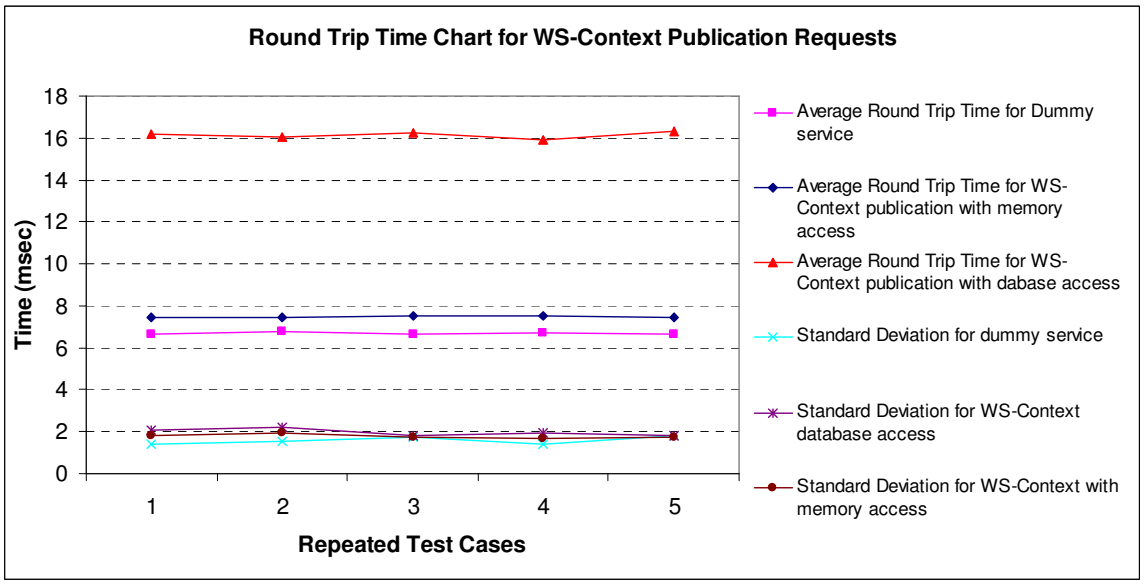

Figure 4 Round Trip Time Chart for Publication Requests

Statistics for the first test set from different publication request testing cases		
	mean	stdev
Test-1	6.64	1.82
Test-2	7.46	1.40
Test-3	16.19	2.06

Table 8 Statistics for the first test set from each testing cases conducted to test publication operation performance. (Test-1: Dummy service testing case, Test-2: WS-Context publication with memory access testing case, Test-3: WS-Context publication with database access testing case.) Time units are in milliseconds.

By comparing the results of inquiry and publication functions when the request is granted with database access, we observe that publication again requires more time compared to inquiry. This performance difference is the effect the database commit that must take place for publication operations.

Secondly, we conducted two testing cases on the system to investigate the scalability to answer following two questions: a) how well does the system perform when the context payload size gets increased, b) how well does the system perform when the message rate per second gets increased.

In the first testing case, our goal is to quantify the degradation in response time when contexts, with larger sizes, published/retrieved into/from the hybrid Service. We have done this by increasing the context sizes until the response time degrades. In this experiment round trip time was recorded at each inquiry/publication request message. The results are depicted in Figure 5 and Figure 6. The detailed statistics corresponding to this experiment is listed in

Table 9 and Table 10.

In the second testing case, we want to determine how well the number of users anticipated can be supported by the system for constant loads. Our goal is to quantify the degradation in response time at various levels of simultaneous users. In order to understand such performance degradation, we evaluate standard hybrid Service functionalities with additional concurrent traffic. We have done this by ramping-up the number of messages sent per second until the system performance degrades. In this testing case, messages were sent of randomly within a second. We again recorded the round trip time at each inquiry/publication request message and applied this test for both publication and inquiry standard operations. The results are depicted in Figure 7. The detailed statistics are given in Table 11.

Based on the results, we note that WS-Context standard operations performed well for small-size context payloads. For example, Figure 5 indicates that the cost of inquiry and publication operations remains the same, as the context's payload size increases from 100Bytes up to 10KBytes. Figure 6 indicates the system behavior for the message sizes between 10Kbytes and 100Kbytes. Based on these results, we observe a linear increase for the time required to complete a publication operation. By comparing the results from a dummy service, which returns the same size message with no processing applied, and hybrid service, we observe that the pure server

processing time remains the same as the size of the messages vary. In our implementation, we keep all available metadata identifiers in memory as well as the modest size metadata values. Thus, if the metadata value is above a certain value, which could be specified in the configuration files, then the system makes a database access to store/retrieve corresponding metadata. This way, the system is able to keep high number of modest size messages in memory. Figure 6 also indicates the system performance behavior when the size of value requires a database access for the context payload sizes ranging from 10 Kbytes to 100 Kbytes. We observe an increase of ~7 ms when the publication operation requires a database access.

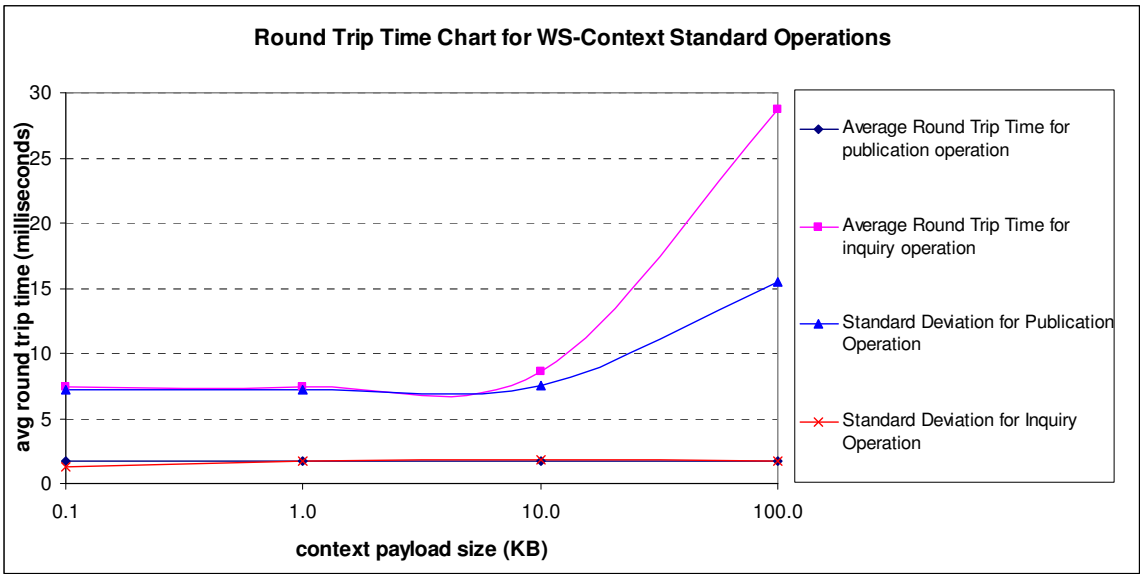

Figure 5 Logarithmic scale round trip time chart for inquiry and publication requests when context payload size increases

	WS-Context inquiry operation		**WS-Context publication operation**	
Kbytes	mean	stdev	mean	stdev
0.1	7.18	1.34	7.38	1.70
1	7.17	1.73	7.43	1.75
10	7.50	1.79	8.58	1.67
100	15.50	1.77	28.76	1.75

Table 9 Statistics of Figure 5 for hybrid Information Service - WS-Context inquiry and publication operations with changing context payload sizes. Time units are in milliseconds.

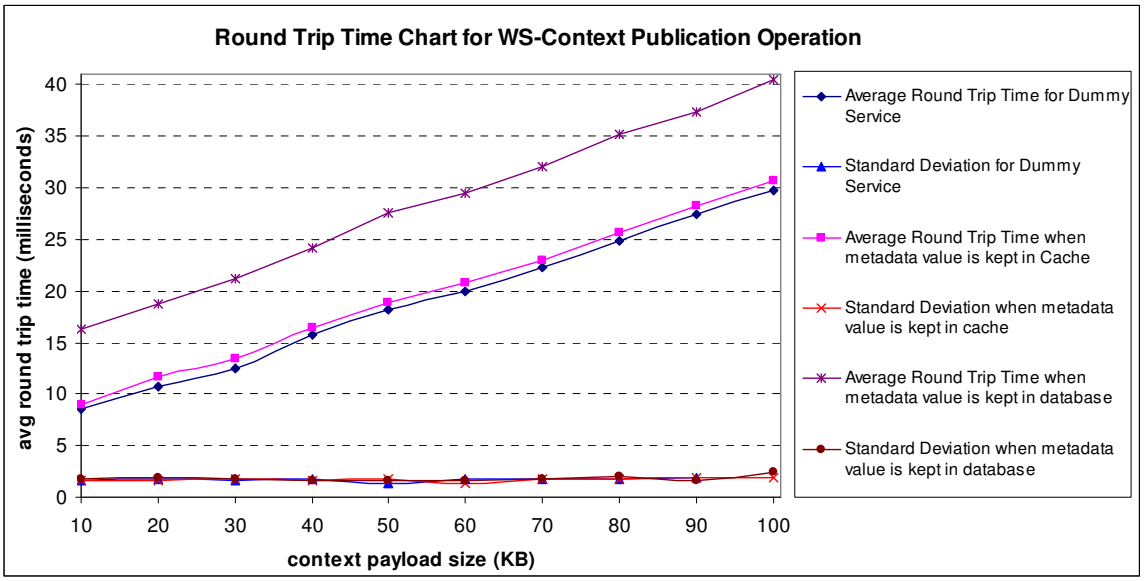

Figure 6 Round Trip Time chart for publication requests when context payload size increases from 10Kbytes to 100Kbtyes

	Dummy service		**Hybrid Service with memory access**		**Hybrid-Service with database access**	
Kbytes	mean	stdev	mean	stdev	mean	stdev
10	8.58	1.67	8.93	1.68	16.33	1.79
20	10.78	1.66	11.68	1.67	18.78	1.86
30	12.52	1.72	13.50	1.74	21.23	1.76
40	15.72	1.67	16.42	1.67	24.12	1.62
50	18.17	1.73	18.87	1.75	27.57	1.65
60	19.94	1.41	20.73	1.40	29.43	1.68
70	22.29	1.76	22.98	1.76	31.98	1.72
80	24.85	1.83	25.70	1.83	35.17	2.05
90	27.38	1.83	28.29	1.84	37.37	1.58
100	29.73	1.94	30.64	1.93	40.51	2.42

Table 10 Statistics of Figure 6 for hybrid service - WS-Context publication operations with changing context payload sizes. Time units are in milliseconds

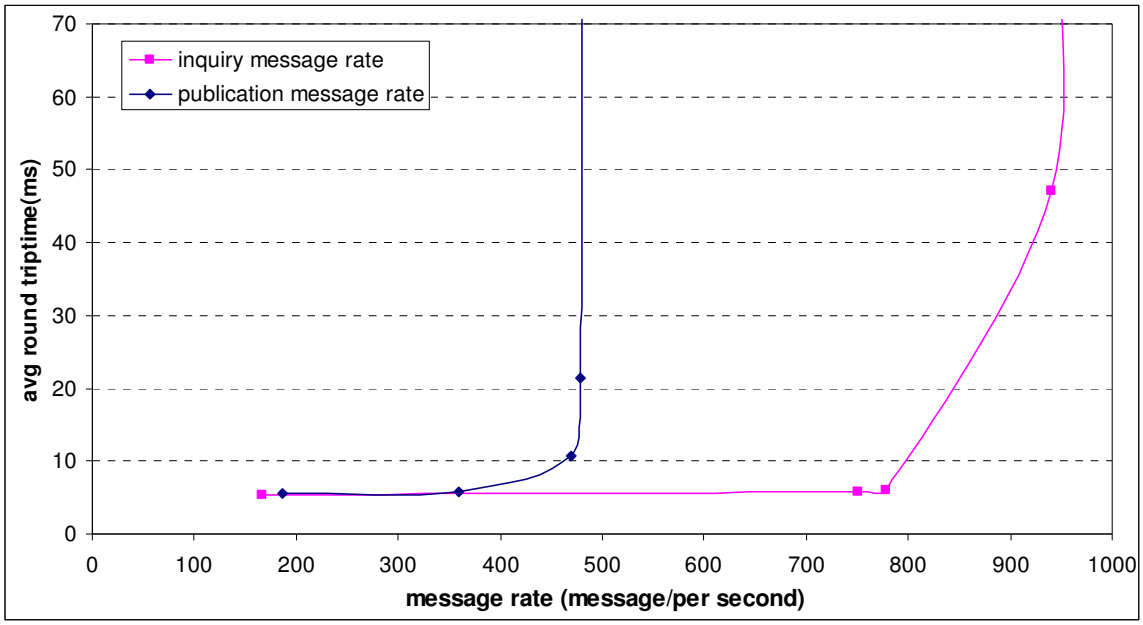

Figure 7 Average hybrid service - WS-Context inquiry and publication response time chart - response time at various levels of message rates per (simultaneous users) per second

WS-Context inquiry operation		
messages/second	mean	stdev
167	5.45	0.65
751	5.84	0.97
778	5.9	0.91
940	47.05	33.52
942	92.25	45.13
WS-Context publication operation		
messages/second	mean	stdev
186	5.65	10.31
359	5.86	39.49
469	10.69	53
479	21.36	203.65
480	70.57	300.56

Table 11 Statistics of the experiment depicted in Figure 7. These measurements were taken with the service when the inquiry and publication request is granted with memory access. Time units are in milliseconds.

Based on the results depicted in Figure 7 and listed in Table 11, we determine that large number of concurrent inquiry requests may well be responded without any error by the system and do not cause significant overhead on the system performance. We observe a threshold between 800 and 940 inquiry messages per second, for the maximum number of concurrent inquiries that can be handled by the system within a second. This threshold is mainly due to the limitations of Web Service container, as we observe the similar threshold when we test the system with an echo service that returns the input parameter passed to it with no message processing is applied.

Based on the results depicted in Figure 7 and listed in Table 11, we also determine that a significant number of concurrent publication requests may well be responded without any error

by the system and do not cause big overhead on the system performance. We observed a threshold, ~ 360 - 480 publication messages per second, for the maximum number of concurrent publication requests that can be handled by the system within a second. This threshold is mainly due to the fault-tolerance level. As the publication message rate is increased, the number of updated/newly written contexts (within a unit time interval) in the ContextSpaces (TupleSpaces) is also increased. In turn, the time required for writing the larger number of updates into MySQL database is increased. Thus, we see higher fluctuations in the response times for increasing number of simultaneous publication requests by examining the standard deviations results listed in Table 11. This experimental study points out the inevitable trade-off between the fault-tolerance and scalability. The lower the fault-tolerance level, the higher the scalability numbers would be for publication operations.

7. Conclusions

We examined the hybrid Grid Information Service as an important tool for knowledge and information grids. We focused on the semantics and identified the base elements of the architecture: data semantics and semantics for XML API sets such as publication, inquiry, security and access control. With this identification made, we discussed our approach and experiences in designing "semantics" for hybrid service. Also, we outlined a real-life application use scenario to identify a way and reason of using hybrid information service in Grids. We implemented the system as open-source software which has been used successfully in varying types of Grids: collaboration, earth science and so forth. We discussed the prototype implementation design and presented the results of the proposed approach.

Acknowledgement: *This work was supported in part by the U.S. National Aeronautical and Space Administration's Advanced Information Systems Technology program. The authors would like to thank Prof. Gordon Erlebacher for his critique on the Hybrid Information Service project and Community Grids Laboratory graduate research assistants who have been using the system in their applications*.

8. References

[1] Zhuge, H., China's E-Science Knowledge Grid Environment, IEEE Intelligent Systems, 19 (1), (2004) 13-17

[2] Bellwood, T., et al. UDDI Version 3.0.1: UDDI Spec Technical Committee Specification. Available from http://uddi.org/pubs/uddi-v3.0.1-20031014.htm.

[3] Bunting, B., et al. K. Web Services Context (WS-Context), available from http://www.arjuna.com/library/-specs/ws_caf_1-0/WS-CTX.pdf

[4] V. Dialani. UDDI-M Version 1.0 API Specification. University of Southampton – UK. 02.

[5] Ali ShaikhAli, el al. UDDIe: An Extended Registry for Web Services. Proc. of the Service Oriented Computing: Models, Architectures and Applications, SAINT-2003 IEEE Comp. Society Press., USA

[6] Czajkowski, K., et al. 2004. The WS-Resource Framework.

http://www.globus.org/wsrf/specs/ws-wsrf.pdf

[7] Alexander, J., et al. 2004 The Web Service Transfer (WS-Transfer) http://msdn.microsoft.com/library/en-us/dnglobspec/html/wstransfer.pdf

[8] Ballinger, K., et al. 2004 The Web Services Metadata Exchange http://specs.xmlsoap.org/ws/2004/09/mex/WS-MetadataExchange.pdf

[9] Aktas, M. S., Fox, G. C., Pierce, M. Fault Tolerant High Performance Information Services for Dynamic Collections of Grid and Web Services FGCS Special issue from SKG2005 Beijing China November, 2005.

[10] Aktas, M. S., Fox, G. C., Pierce, M., Managing Dynamic Metadata as Context. The 2005 Istanbul International Computational Science and Engineering Conference (ICCSE2005), Istanbul, Turkey.

[11] Oh, S., Aktas, M. S., Pierce, M., Fox, G. C., Optimizing Web Service Messaging Performance Using a Context Store for Static Data, 5th WSEAS Int. Conf. on Telecommunications and Informatics, Turkey, 05

[12] Gelernter, N. C. a. D. (1989). "Linda in Context." Commun. ACM, 32(4): 444-458.

[13] Coleman, R., et al., MicroSpaces software with version 1.5.2 available at http://microspaces.sourceforge.net/. May 2004.

[14] Fox, S. P. a. G. (2003). NaradaBrokering: A Distributed Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids. Proceedings of ACM/IFIP/USENIX International Middleware Conference Middleware-2003, Rio Janeiro, Brazil.