Information Federation in Grid Information Services

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

[1] Community Grids Laboratory, Indiana University
501 N. Morton Suite 224, Bloomington, IN 47404
{maktas, gcf, mpierce}@cs.indiana.edu
www.communitygrids.iu.edu
[2] Computer Science Department, School of Informatics, Indiana University
[3] Physics Department, College of Arts and Sciences, Indiana University

Abstract. Independent Grid projects have developed their own solutions to Information Services. These solutions are not interoperable with each other, target vastly different systems and address diverse sets of requirements. To address these challenges, we designed a novel architecture for a Grid Information Service that provides unification, federation and interoperability of major grid information services. The proposed approach forms an add-on information system that interacts with the local information services and assembles their metadata instances under one hybrid architecture. We present our experiences in designing semantics and architectural design of this system. We introduce a prototype implementation and present its evaluation. The results indicate that the proposed approach achieves unification and federation of custom implementations of grid information services with negligible processing overheads.

1 Introduction

Independent Grid projects have developed their own solutions to Information Services. These solutions are not interoperable with each other, target vastly different systems and address diverse sets of requirements. For an example, large-scale Grid applications require management of large amounts of relatively slowly varying metadata. Another example, e-Science Grid applications may be thought of as dynamically assembled collections of modest numbers of distributed services that are assembled for specific tasks that can be as diverse as forecasting earthquakes [1] or managing audiovisual collaboration sessions [2]. These dynamic Grid/Web service collections require greater support for dynamic metadata.

Existing solutions to Grid Information Services present some challenges: Firstly, independent Grid applications use customized implementations of Grid Information Services whose data model and communication language is different [3]. These information services require greater interperoperability to enable communication between different grid projects so that they can share and utilize each other's resources [4]. Secondly, previous solutions do not address metadata management requirements of most Grid applications that have both large-scale, static and small-scale, highly-dynamic metadata associated to Grid/Web Services [3]. Thirdly, existing solutions do not provide uniform interfaces for publishing and discovery of both dynamically generated and static information [3]. These solutions create a limitation on the client-end, as the users have to interact with more than one metadata service. This in turn increases the complexity of clients and creates fat clients. We therefore see this as an important area of investigation.

To address these challenges, an ideal Grid Information Service Architecture should meet the following requirements: uniformity, federation, interoperability, performance, persistency and fault-tolerance. Uniformity: To meet the uniformity requirement, it should support one-to-many information services and their communication protocols. Federation: To meet the federation requirement, it should present a federation capability where different information services can interoperate with each other. Interoperability: To meet the interoperability requirement, it should be compatible with widely-used, existing Grid/Web Service standards. Performance: To meet the performance requirement, it should search/access/store metadata with negligible processing overheads. Persistency: To meet the persistency requirement, it should back-up metadata without degradation of the system performance. Fault-tolerance: To meet the fault-tolerance requirement, it should achieve distribution and redundancy of information.

We propose a Grid Information Service Architecture called Hybrid Grid Information Service (Hybrid Service) that meets the aforementioned requirements. The Hybrid Service enables unification, federation and interoperability of major Grid Information Services. It is designed as add-on system that runs one layer above the existing information service implementations. It improves the capabilities of customized implementations of information service specifications in terms of high-performance and fault-tolerance.

To achieve unification, the Hybrid Service is designed as a generic system, so that it can support one-to-many information service implementations as local data sources. To achieve federation, the Hybrid Service is designed to integrate all kinds of Grid/Web Service metadata at a higher conceptual level, while ignoring the implementation details of the local data-systems. To achieve interoperability, the Hybrid Service is provided with the implementations two widely-used Grid/Web Service Specifications: the WS-I compatible Web Service Context (WS-Context) [5] and Universal Description, Discovery and Integration (UDDI) [6] Specifications. The WS-Context XML Metadata Service implements the WS-Context Specification and manages session-related, interaction-dependent metadata associated to Grid/Web Services. It presents a schema and XML API for the Context Manager component of the WS-Context Specification. It expands on the existing WS-Context Specification and provides advanced capabilities such as a) support for real-time replay capabilities (in particular for collaboration domain), b) support for session failure recovery, and c) parent-child relationships on the state information of the Grid/Web Services. The extended UDDI XML Metadata Service implements an extended version of UDDI Specification. It manages interaction-independent, rarely changing metadata associated to Grid/Web Services. It supports static metadata management requirements of Grid/Web Services. It is designed to be a domain-independent metadata service to meet with the static, stateless information requirements of the targeted application use domain. To meet with the specific metadata requirements of Geographical Information Systems, the design was further extended to support geospatial queries on the metadata catalog. This service introduces various capabilities: a) publishing additional metadata associated with service entries, b) posing metadata-oriented, geospatial, and domain-independent queries on the static metadata catalog and c) aggregating and searching geospatial services. To achieve high-performance, the Hybrid Service is designed to employ an in-memory storage capability. This in turn minimizes the processing overheads for metadata access/storage. To achieve fault-tolerance, the Hybrid Service is designed as a decentralized system in which the network nodes communicate through publish-subscribe based messaging schemes.

In this paper, we present our experiences in designing semantics and architectural design of the Hybrid Service. We introduce a prototype implementation of this architecture and present its performance evaluation. As the main focus of this paper is information federation in Grid Information Services, we discuss unification, federation, interoperability and performance aspects of the system and leave out distribution and fault-tolerance aspects as discussion of another paper [7].

This section presented a general introduction of the proposed research. First, the limitations in existing Grid Information Service solutions, which lead into the proposed research, were discussed. Then, to present the scope of the research, the requirements expected from this research are outlined. Next, a brief summary of the proposed system is discussed. The organization of the rest of the paper is as follows. Section 2 reviews the relevant work in state of art of the studies covered in this paper. Section 3 gives an overview of the system. Section 4 presents the semantics of the Hybrid Service. Section 5 presents the architectural design details and the prototype implementation of the system. Section 6 analyzes the performance evolution of the Hybrid Service prototype. It presents benchmarking on performance and scalability aspects of the system. Section 7 contains the summary and the future research directions.

2 Relevant Work

Information integration is the process of unifying information residing at multiple sources and providing a unified access interface [8]. Unifying heterogeneous data sources under a single architecture has been target of many investigations [9]. For example, information integration research area is mainly studied by distributed database systems research [10, 11]. It investigates how to share data at a higher conceptual level, while ignoring the implementation details of the local data systems. In turn, this effort enables transparent access to multiple, logically interrelated distributed databases. Based on this scheme, an application can pose a query to the distributed database system, which maps the query into local queries, integrates the results coming from different data systems and return the results to the client. Previous work on such merger between the heterogeneous information systems can be broadly categorized as global-as-view and local-as-view integration [12]. In former category, data from several sources are transformed into a global schema and can be queried with a uniform query interface. In the latter category, queries are transformed into specialized queries over the local databases. In this category, integration is carried out by transforming queries. **Limitations:** The global schema approach captures expressiveness capabilities of customized local schemas. However, this approach cannot scale up to high number of data sources. Another drawback is the need to update the global schema whenever a new schema is to be integrated and/or an existing local system changes its schema. In the local-as-view approach, because of the lack of a global schema in the data integration architecture, each local-system's schema may need to be mapped against each other. This in turn will lead to large number of mappings that need to be created and managed. **Discussion:** To achieve data integration, global-as-view or local-as-view approaches can be utilized. In the local-as-view approach, information integration happens through query processing. In other words, this approach transforms the client's query into local queries and integrates the results. This methodology has performance drawbacks due to overhead of query mapping and forwarding. Furthermore, in architectures such as those of federated database systems, a high number of query mappings may be required. To achieve high performance, there is a need for a higher-level add-on architecture that can assemble the information coming from different metadata systems and carry out queries on the heterogeneous information space. This approach should be designed in such a way that the single repository should be distributed to avoid single point of failure. We think that once we achieve such higher-level architecture, the global-as-view approach can be used for integrating heterogeneous local information services. This approach encapsulates the expressiveness power of the customized schemas that are being integrated. In this research, we design and build an architecture of a Grid Information Service that would support information integration. To achieve this objective, we revisit the research ideas in distributed database systems and utilize global-as-view approach in our architecture. To sum up, we take as a design requirement that the proposed system should be designed as an add-on architecture above existing Grid Information Services to provide unification and federation of information coming from different metadata systems.

Efforts towards interoperability in Grid Community has recently been promoted by the Open Grid Forum (OGF). The OGF [13] has started a research activity called GIN (Grid Interoperation Now) [4] to manage interoperation among major grid projects such as EGEE [14], UK National Grid Service [15], NorduGrid [16]. This effort includes interoperation in the areas of authorization and identity management, data management and movement, job description and submission, information services and schema, and operations experience of pilot test applications. Among these interoperation efforts, interoperability of information services is also the focus of this research. The OGF suggests guidelines for interoperability in such a way that each grid's internal information system will act as a translator for accessing information from other information services. In this scenario, all of grid information can be retrieved by each of grid in its fashion with respect to schema and query language. As the information service schema, the Open Grid Forum GIN workgroup utilizes a subset of the Glue schema as the common description schema for information services. The Grid Laboratory Uniform Environment (Glue) Schema [17] is an effort to support interoperability between US and Europe Grid Projects. It presents description of core Grid resources at the conceptual level by defining an information model. It is used for both monitoring and discovery purposes and describes the state and functionalities of Grid resources. **Discussion:** In this research, we propose a system architecture that meets the interoperability guidelines suggested by OGF GIN work group. To this end, we integrate the Glue Schema into our design to be able to interoperate with GIN activity participating information services. With this study, we also intend to build an architecture that would address wide range of Web Service applications and provide an interoperation-bridge across the existing implementations of information services. Thus, we implement two widely used and WS-I compatible grid information services: Extended UDDI XML Metadata Service and WS-Context XML Metadata Service.

The Universal Description, Discovery, and Integration (UDDI) Specification is a widely used standard that enables services advertise themselves and discover other services. It is a WS-Interoperability (WS-I) compatible standard. It is designed as domain-independent, standardized method for publishing/discovering information about Web Services. It offers users a unified and systematic way to find service providers through a centralized registry of services. A number of studies extends and improves the out-of-box UDDI Specification. For an example, Open Geographical Information Systems Consortium [18] introduced a set of design principles, requirements and spatial discovery methodologies for discovery of OGC services through UDDI interface [19]. These methodologies have been implemented by various organizations such as Sycline [20] and Galdos [21]. The Syncline experiment focuses on implementing a UDDI discovery interface on an existing OGC Catalog Service data model so that UDDI users can discover services registered through OGC Registries. The Galdos experiment focuses on turning OGC Service Registry into a UDDI node by utilizing JAXR API to map UDDI inquiry interface to the OGC Registry Information Model. These methodologies showed that it is possible to do spatial discovery and content discovery through UDDI Specification. Other examples, UDDI-M [22] and UDDIe [23] projects introduced 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. METEOR-S [24] leveraged UDDI Specification by utilizing semantic web languages and identifying different semantics when describing a service, such as data, functional, quality of service and executions. Grimories [25] extends the functionalities of UDDI to provide a semantic enabled registry designed and developed for the MyGrid project [26]. It supports third-party attachment of metadata about services and represents all published metadata in the form of RDF triples either in a database, or in a file, or in a memory. **Limitations:** We find following limitations in the existing out-of-box UDDI specifications: Firstly, UDDI introduces keyword-based retrieval mechanism and does not allow advanced metadata-oriented query capabilities. Secondly, UDDI does not take into account the volatile behavior of services. Thirdly, it does not provide domain-specific query capabilities such as geospatial queries. We find the following limitations in the OGC's UDDI approach: Firstly, it is designed for and limited to geospatial specific usage. Secondly, OGC approach does not define a data model rich enough to capture descriptive metadata that might be associated with service entries. We also find limitations in the existing UDDI-Extensions: These approaches have investigated a generic and centralized metadata service focusing on the domain-independent metadata management problems. However, these solutions, as they are generic, do not solve the domain-specific metadata management problems as we see in geographical information system domain. **Discussion:** The UDDI Specification is promising as being a widely used WS-I compatible standard to manage semi-static metadata associated to Web Services. For this research, we built a UDDI XML Metadata Service addressing the aforementioned limitations of previous UDDI solutions. This implementation manages both prescriptive and descriptive metadata associated to Grid/Web Services and addresses metadata management requirements of geospatial services.

The Web Services Context (WS-Context) Specification [5] defines a simple mechanism to share and keep track of common information shared between multiple participants in Web Service interactions. It is a lightweight storage mechanism, which allows the participant's of an activity to propagate and share context information. It defines an activity as a unit of distributed work involving one or more parties (services, components). In order for an activity to extend over a number of Web Services, certain information has to flow among the participant of application. This specification refers such information as context and focuses on its management. The WS-Context Specification defines three main components: a) context service, b) context, and c) an activity lifecycle service. The context service is the core service concerned with managing lifecycle of context propagation. The context defines information about an activity and is referenced with a URI. It allows a collection of actions to take place for a common outcome. The minimum required context information (such as the context URI) is exchanged among Web Services in the header of SOAP messages to correlate the distributed work in an activity. This way, a participant service obtains the identifier and makes a key-based retrieval on the context service. Thus, a typical search with the WS-Context is mainly based on key-based retrieval/publication capabilities. The activity of lifecycle service defines the scope of a component activity. Note that, activities can be nested. An activity may be a component activity of another. In this case, additional information (such as security metadata) to a basic context may be kept in a component service, which is registered with the core context service and participate in the lifecycle of an activity. **Limitations:** We find following limitations in WS-Context Specification. Firstly, the context service, a component defined by WS-Context to provide access/storage to state information, has limited functionalities such as the two primary operations: GetContext and SetContext. However, traditional and Semantic Grid applications present extensive metadata needs which in turn requires advanced search/access/store interface to distributed session state information. Secondly, the WS-Context Specification is only focused on defining stateful interactions of Web Services. It does not define a searchable repository for interaction-independent information associated to the services involved in an activity. However, there is a need for a unified specification, which can provide an interface not only for stateful metadata but also for the stateless, interaction-independent metadata associated to Web Services. **Discussion:** The WS-Context Specification is a promising approach to tackle the problem of managing distributed session state, since it models a session metadata repository as an external entity where more than two services can easily access/store highly dynamic, shared metadata. For this research, we implemented a prototype of the WS-Context – Context Manager Service by expanding on the out-of-box WS-Context Specifications. This implementation manages dynamically generated session-related metadata.

Information security is a fundamental issue in Grid Information Services, as the Grid/Web Service metadata may not be open to anyone. Thus, there is a need for an information security mechanism. Managing information security deals with managing access rights. By reviewing the previous work, we observe that the capability-based access control [27] is a commonly used approach for managing access rights. It is used to give each user a list of capabilities to give the access rights related with the metadata. In this scenario, a user can only access the metadata if he/she has sufficient access rights [28]. The use of a protection domain is another approach in which the system grants the request and carries out the operation by first checking with the protection domain associated with that request [29]. A protection domain can also be a role. In this case, the role of user determines the protection domain of that user and the system grants the incoming request based on the user's role [30]. **Discussion:** In this study, we leave out the investigating and leveraging of research in the information security area as future work. We concentrate on the unification, federation and interoperability aspects of the system.

TupleSpaces is an associated memory paradigm. A TupleSpace forms an associated shared memory through which two or more processes can exchange/share data. It provides mutual exclusive access, associative lookup and persistence for a repository of tuples that can be accessed concurrently. Thus, a tuplespace can be used to coordinate events of processes. A tuplespace is comprised of a set of tuples: data structures containing typed fields where each field contains a value. A small example of a tuple would be: ("context_id", Context), which indicates a tuple with two fields: a) a string, "context_id" and b) an object, "Context". The tuplespace was first introduced by Gelernter and Carriero at Yale University [31] as a part of Linda programming language. Linda consists fundamentally of four operations ("in", "rd", "out" and "eval") through which tuples can be added, retrieved or taken from a tuplespace. The JavaSpaces [32] project by Sun extends and implements Linda. Likewise, IBM has a tuplespaces implementation called TSpaces [33]. Linda has been extended to support different types of communication and coordination between systems and has increased some interest in diverse communities such as the ubiquitous computing (sTuples [34]) and Semantic Web (Triple Spaces [35], Semantic Web Spaces [36]). **Discussion:** The tuplespaces paradigm provides mutually exclusive access, which in turn enables data sharing between processes. This way both the shared memory and the processes are temporarily and spatially uncoupled. We take as a requirement that our design should employ the tuplespaces paradigm as an in-memory storage to meet aforementioned performance requirement of the system. A java implementation of the TupleSpaces concept, JavaSpaces [32], was released by Sun MicroSystems . However, JavaSpaces requires a number of daemon services to run including a naming service, a restart service, and the JavaSpaces service. These services add complexity to the systems employing JavaSpaces. MicroSpaces [37], an open-source implementation of TupleSpaces paradigm, is an alternative collection of java libraries and provides same API semantics identical with JavaSpaces. MicroSpaces is a multi-threaded application and dependent on RMI to provide interactions with JavaSpaces. Apart from the existing implementation approaches, we take as a requirement that our design should support a lightweight implementation of JavaSpaces that does not require RMI-based communication protocol or other daemon services to run.

3 Hybrid Service

We designed and built a novel Grid Information Service Architecture called Hybrid Grid Information Service (Hybrid Service) which provides unification, federation and interoperability of Grid Information Services. The Hybrid Service forms an add-on architecture that interacts with the local information services and unifies them in a higher-level hybrid system. In other words, it provides a unifying architecture where one can assemble metadata instances of different information services. To achieve this, the Hybrid System Architecture introduces abstraction layers such as uniform access interface and information resource management as illustrated in Figure 1. The uniform access layer is used to support one-to-many communication protocols. The information resource management layer is used to manage one-to-many local schema implementations.

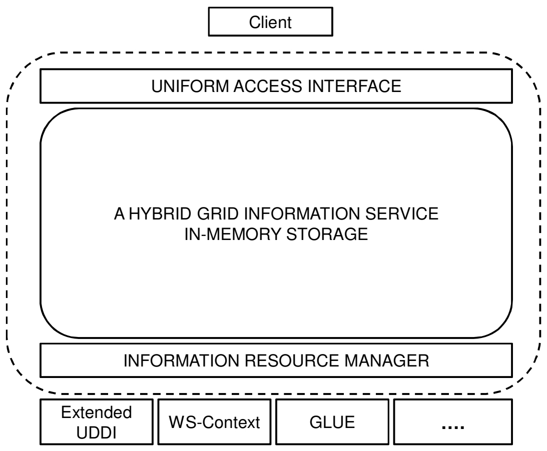


Figure 1 A general view of the Hybrid Service rounded with a dashed box.

In our prototype implementation, we showed that the Hybrid Service is able to achieve unification of the two local information service implementations: WS-Context and Extended UDDI and support their communication protocols. We also showed that the Hybrid Service is able to achieve information federation by utilizing a global schema, which integrates local information service schemas, and user-provided mapping rules, which provides transformations between the metadata instances of the global schema and the local schemas. With these capabilities, the Hybrid Services enables different Grid Information Service implementations to interact with each other and share each other's metadata. Furthermore, the Hybrid Service provides the ability of issuing integrated queries on the heterogeneous metadata space where metadata comes from different information service providers. In turn, this enables the system to support an integrated access to not only quasi-static, rarely changing interaction-independent metadata, but also highly updated, dynamic interaction-dependent metadata associated to Grid/Web Services. We discuss semantics of the Hybrid Service in the following section followed by a section discussing the architecture of the system.

4 Semantics

In this section, first, we discuss four information service specifications: the Extended UDDI, the WS-Context, the Glue Schema and the Unified Schema Specifications. The extended UDDI Specification extends existing out-of-box UDDI Specification to address its aforementioned limitations (see Section 2 for relevant discussion). The WS-Context Specification improves the existing out-of-box Web-Service Context Specification to meet the aforementioned requirements of the Hybrid Service (see Section 1 for relevant discussion). The Glue Schema Specification is used as-is to be able to support interoperability with the US and Europe Grid projects. Finally, the Unified Schema Specification integrates these three information service specifications. Then, we discuss the two Hybrid Service Schemas: Hybrid Schema and SpecMetadata Schema, which define the necessary abstract data models to achieve a generic architecture for unification and federation of different information service implementations in the Hybrid Service.

4.1. The Extended UDDI Specification

We have designed extensions to the out-of-box UDDI Data Structure (described in [6]) to be able to associate both prescriptive and descriptive metadata with service entries. This way the system can interoperate with existing UDDI clients without requiring an excessive change in the implementations. UDDI-M [22] and UDDIe [23] projects introduced the idea of associating simple (name, value) pairs with service entities. This methodology is promising as it provides a generic metadata catalog and yet it has its own merits of simplicity in implementation. Thus, we adopt this approach and expand on existing UDDI Specifications as described in the following section. We briefly discuss an earlier version of our extended UDDI Service in [38, 39].

<u>Extended UDDI Schema:</u> We introduced an extended UDDI data model (see Figure 2) to address the metadata requirements of Geographical Information System/Sensor Grids. The existing UDDI Data Model consists of following core entities: businessEntity, businessService, bindingTemplate, publisherAssertions and tModel. A businessEntity contains information about the party who publishes information about a service. It may contain one-to-many businessService entities. The publisherAssertions entity defines the relationship between the two businessEntities. A businessService entity provides descriptive information about a particular family of Grid/Web Services. It may contain one-to-many bindingTemplate entities, which define the technical information about a service end-point. A bindingTemplate entity contains references to tModel, which defines descriptions of specifications for service end-points.

In our approach, we expanded on the out-of-box UDDI data model. This data model includes following additional/modified entities: a) service attribute entity (serviceAttribute) and b) extended business service entity (businessService). Here, each businessService entity is associated with one-to-many serviceAttribute entities. We describe the additional/modified data model entities (both the serviceAttribute and businessService entities) in the next sections.

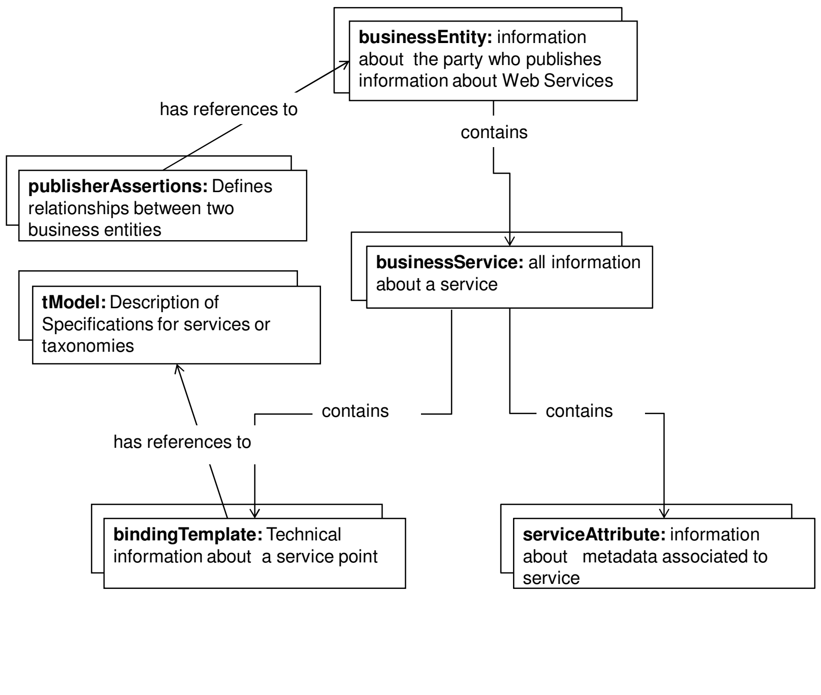

businessEntity: information about the party who publishes information about Web Services
has references to
contains
publisherAssertions: Defines relationships between two business entities
businessService: all information about a service
tModel: Description of Specifications for services or taxonomies
contains
contains
has references to
bindingTemplate: Technical information about a service point
serviceAttribute: information about metadata associated to service

Figure 2 Extended UDDI Service Schema

Business service entity structure: The UDDI's business service entity structure contains descriptive, yet limited information about Web Services. A comprehensive description of the out-of-box business service entity structure defined by UDDI can be found in [6]. Here, we only discuss the additional XML structures introduced to expand on existing business service entity. (The structure diagram for business service entity is illustrated in Figure 3)

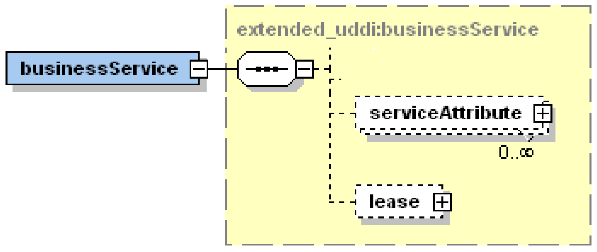


Figure 3 Partial structure diagram for businessService entity

These additional XML elements are a) service attribute and b) lease. The service attribute XML element corresponds to a static metadata (e.g. WSDL of a given service). Similar to session entity, a business service entity may have a lifetime associated with it. A lease structure describes a period of time during which a service can be discoverable.

Service attribute entity structure: A service attribute (serviceAttribute) data structure describes information associated with service entities. The structure diagram for serviceAttribute entity is illustrated in Figure 4. Each service attribute corresponds to a piece of metadata, and it is simply expressed with (name, value) pairs. Apart from similar approaches [22, 23], in the proposed system, a service attribute includes a) a list of abstractAtttributeData, b) a categoryBag and c) a boundingBox XML structures. An abstractAttributeData element is used to represent metadata that is directly related with functionality of the service and store/maintain these domain specific auxiliary files as-is. This allows us to add third-party data models such as "capabilities.xml" metadata file describing the data coverage of domain-specific services such as the geospatial services. An abstractAttributeData can be in any representation format such as XML or RDF. This data structure allows us to pose domain-specific queries on the metadata catalog. Say, an abstractAttributeData of a geospatial service entry contains "capabilities.xml" metadata file. As it is in XML format, a client may conduct a find_service operation with an XPATH query statement to be carried out on the abstractAttributeData, i.e. "capabilities.xml". In this case, the results will be the list of geospatial service entries that satisfy the domain-specific XPATH query.

The categoryBag is used to provide a custom classification scheme to categorize serviceAttribute elements. A simple classification could be whether the service attribute is prescriptive or descriptive. A boundingBox element is used to describe both temporal and spatial attributes of a given geographic feature. This way the system enables spatial query capabilities on the metadata catalog.

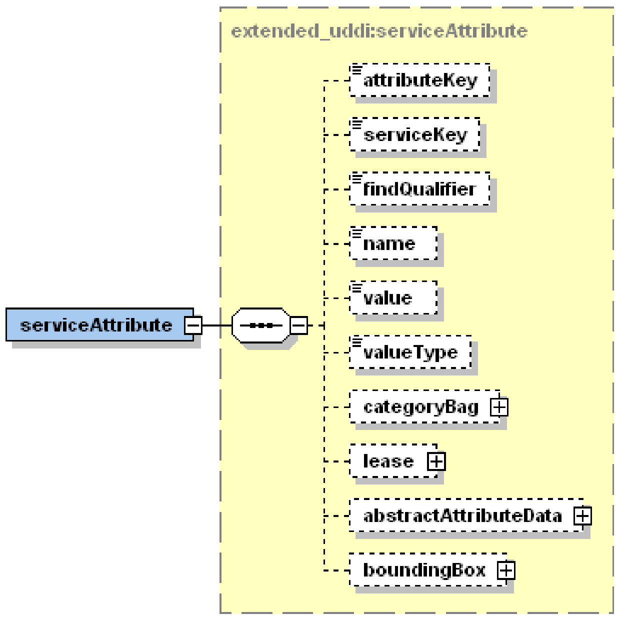


Figure 4 Structure diagram for serviceAttribute

Extended UDDI Schema XML API: We present extensions/modifications to existing UDDI XML API set to standardize the additional capabilities of our implementation. These additional capabilities can be grouped under two XML API categories: Publish and Inquiry.

Function	Category	Information Service
Save_service	Publish	Extended UDDI API: This API is to support/handle interaction-independent metadata associated to services.
Save_serviceAttribute		
Delete_service		
Delete_serviceAttribute		
Get_serviceDetail	Inquiry	
Get_serviceAttributeDetail		
Find_service		
Find_serviceAttribute		

Table 1 The Publish/Inquiry XML API for the extended UDDI Service. The extended UDDI XML API is introduced as an extension to existing UDDI Specification XML API Sets.

Table 1 gives the list of additional XML API that we introduce with the Extended UDDI Service. The Publish XML API is used to publish metadata instances belonging to different entities of the extended UDDI Schema. It extends existing UDDI Publish XML API Set. It consists of the following functions: **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 may have a lifetime (lease). **save serviceAttribute:** Used to register or update one or more semi-static metadata associated with a Web Service. **delete service:** Used to delete one or more service entity structures. **delete serviceAttribute:** Used to delete existing serviceAttribute elements from the service. The Inquiry XML API is used to pose inquiries and to retrieve metadata from the Extended UDDI Information Service. It extends existing UDDI Inquiry XML API set. It consists of the following functions: **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 and Lease to facilitate the additional capabilities. **find serviceAttribute:** Used to find the 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 with a unique identifier. The get serviceAttributeDetail API call returns the serviceAttribute structure corresponding to each of the attributeKey values specified in the arguments. **get serviceDetail:** Used to retrieve service entity structure associated with a unique identifier.

Using Extended UDDI Schema XML API: Given the capabilities of the Extended-UDDI Service, one can simply populate metadata instances using the Extended-UDDI XML API as in the following scenario. Say, a user publishes a new metadata to be attached to an already existing service in the system. In this case, the user constructs a serviceAttribute element. Based on aforementioned extended UDDI data model, each service entry is associated with one or more serviceAttribute XML elements. A serviceAttribute corresponds to a piece of interaction-independent metadata and it is simply expressed with (name, value) pair. We can illustrate a serviceAttribute as in the following example: ((throughput, 0.9)). A serviceAttribute can be associated with a lifetime and categorized based on custom classification schemes. A simple classification could be whether the serviceAttribute is prescriptive or descriptive. In the aforementioned example, the throughput service attribute can be classified as descriptive. In some cases, a serviceAttribute may correspond to a domain-specific metadata where service metadata could be directly related with functionality of the service. For instance, OGC compatible Geographical Information System services provide a "capabilities.xml" metadata file describing the data coverage of geospatial services. We use an abstractAttributeData element to represent such metadata and store/maintain these domain specific auxiliary files as-is. As the serviceAttribute is constructed, it can then be published to the Hybrid Service by using "save_serviceAttribute" operation of the extended UDDI XML API. On receiving a metadata publish request, the system extracts the instance of the serviceAttribute entity from the incoming requests, assigns a unique identifier to it and stores in in-memory storage. Once the publish operation is completed, a response is sent to the publishing client.

4.2. The WS-Context Specification

WS-Context tackles the problem of managing distributed session state. Unlike the point-to-point approaches, WS-Context models a third-party metadata repository as an external entity where more than two services can easily access/store highly dynamic, shared metadata. We investigated semantics for a XML Metadata Service that would expand on WS-Context approach for managing distributed session state information. We discuss our earlier versions of WS-Context Service in [38, 39].

WS-Context Schema: We introduced an information model comprised of following entities: sessionEntity, sessionService and context. Figure 5 illustrates the data model for the WS-Context Service. A sessionEntity describes information about a session under which a service activity takes place. A sessionEntity may contain one-to-many sessionService entities. A sessionService entity describes information about a Web Service participating to a session. Both sessionEntity and sessionService may contain one-to-many context entities. A context entity contains information about interaction-dependent, dynamic metadata associated to either sessionService or sessionEntity or both. Each entity represents specific types of metadata. Instances of these structures have system-defined unique identifiers. An instance of an entity gets its identifier when it is first published into the system. All entities have a lifetime during which the entity instances are expected to be up-to-date. The following sections discuss the core entities of the WS-Context Service Schema.

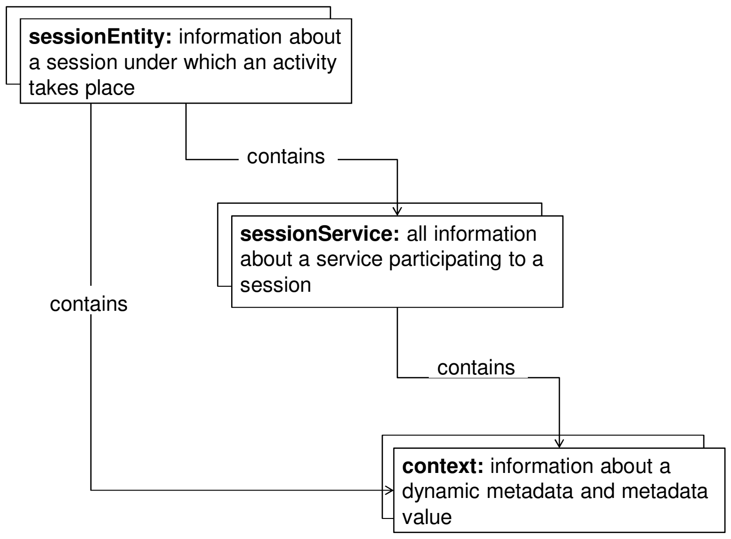

sessionEntity: information about a session under which an activity takes place
contains
sessionService: all information about a service participating to a session
contains
contains
context: information about a dynamic metadata and metadata value

Figure 5 WS-Context Service Schema

Session entity structure: A sessionEntity describes a period of time devoted to a specific activity, associated contexts, and serviceService involved in the activity. A sessionEntity can be considered as an information holder for the dynamically generated information. The structure diagram for sessionEntity is illustrated in Figure 6. An instance of a sessionEntity is uniquely identified with a session key. A session key is generated by the system when an instance of the entity is published. If the session key is specified in a publication operation, the system updates the corresponding entry with the new information. When retrieving an instance of a session, a session key must be presented. A sessionEntity may have name and description associated with it. A name is a user-defined identifier and its uniqueness is up to the session publisher.

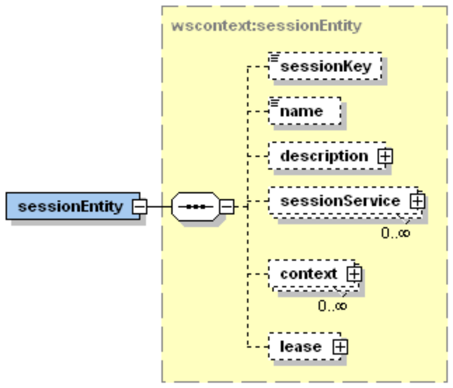


Figure 6 Structure diagram for sessionEntity

A user-defined identifier is useful for the information providers to manage their own data. A description is optional textual information about a session. Each sessionEntity contains one-to-many context entity structures. The context entity structure contains dynamic metadata associated to a Web Service or a session instance or both. Each sessionEntity is associated with its participant sessionServices. The sessionService entity structure is used as an information container for holding limited metadata about a Web Service participating to a session. A lease structure describes a period of time during which a sessionEntity or serviceService or a context entity instances can be discoverable.

Session service entity structure: The sessionService entity contains descriptive, yet limited information about Web Services participating to a session. The structure diagram for sessionService entity is illustrated in Figure 7. A service key identifies a sessionService entity. A sessionService may participate one or more sessions. There is no limit on the number of sessions in which a service can participate. These sessions are identified by session keys. Each sessionService has a name and description associated with it. This entity has an endpoint address field, which describes the endpoint address of the sessionService. Each sessionService may have one or more context entities associated to it. The lease structure identifies the lifetime of the sessionService under consideration.

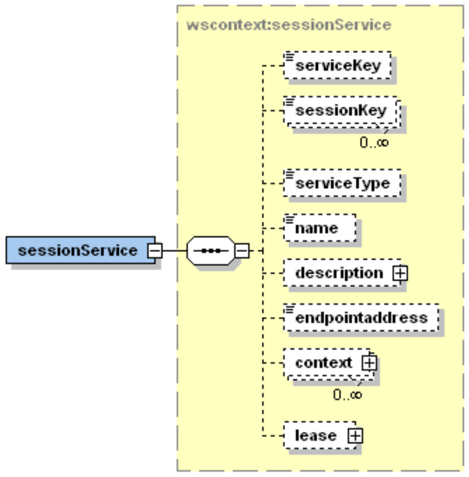


Figure 7 Structure diagram for sessionService

Context entity structure: A context entity describes dynamically generated metadata. The structure diagram for context entity is illustrated in Figure 8. An instance of a context entity is uniquely identified with a context key, which is generated by the system when an instance of the entity is published. If the context key is specified in a publication operation, the system updates the corresponding entry with the new information. When retrieving an instance of a context, a context key must be presented.

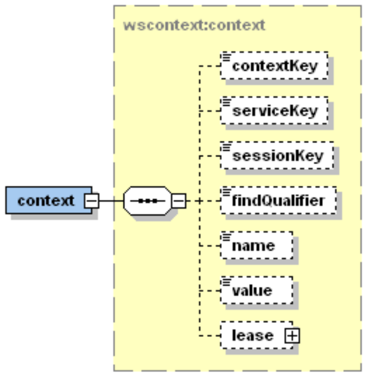


Figure 8 Structure diagram for context entity

A context is associated with a sessionEntity. The session key element uniquely identifies the sessionEntity that is an information container for the context under consideration. A context has also a service key, since it may also be associated with a sessionService participating a session. A context has a name associated with it. A name is a user-defined identifier and its uniqueness is up to the context publisher. The information providers manage their own data in the interaction-dependent context space by using this user-defined identifier. The context value can be in any representation format such as binary, XML or RDF. Each context has a lifetime. Thus, each context entity contains the aforementioned lease structure describing the period of time during which it can be discoverable.

WS-Context Schema XML API: We present an XML API for the WS-Context Service. The XML API sets of the WS-Context XML Metadata Service can be grouped as Publish, Inquiry, Proprietary, and Security.

Function	Category	Information Service
Save_session	Publish Functions	The WS-Context Information Service XML API: This API is to support/handle interaction-dependent metadata associated to both services and sessions.
Save_context		
Save_sessionService		
Delete_session		
Delete_context		
Delete_sessionService		
Get_sessionDetail	Inquiry Functions	
Get_contextDetail		
Get_sessionServiceDetail		
Find_session		
Find_context		
Find_sessionService		
Save_publisher	Proprietary Functions	
Get_publisherDetail		
Delete_publisher		
Find_publisher		
Get_authToken	Security Functions	
Discard_authToken		

Table 2 XML API for the WS-Context Service

Table 2 gives the list of available XML API, which we introduce with the WS-Context Service. The Publish XML API is used to publish metadata instances belonging to different entities of the WS-Context Schema. It extends the WS-Context Specification Publication XML API set. It consists of the following functions: **save session:** Used to add/update one or more session entities into the hybrid service. Each session may contain one-to-many context entity, have a lifetime (lease), and be associated with service entries. **save context:** Used to add/update one or more context (dynamic metadata) entities into the service. **save sessionService:** Used to add/update one or more session service entities into the hybrid service. Each session service may contain one-to-many context entity and have a lifetime (lease). **delete session:** Used to delete one or more sessionEntity structures. **delete context:** Used to delete one or more contextEntity structures. **delete sessionService:** Used to delete one or more session service structures. The Inquiry XML API is used to pose inquiries and to retrieve metadata from service. It extends the existing WS-Context XML API. The extensions to the WS-Context Inquiry API set are outlined as follows: **find session:** Used to find sessionEntity elements. The find session API call returns a session list matching the conditions 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. **find sessionService:** Used to find session service entity elements. The find sessionService API call returns a service list matching the criteria specified in the arguments. **get sessionDetail:** Used to retrieve sessionEntity data structure corresponding to each of the session key values specified in the arguments. **get contextDetail:** Used to retrieve the context structure corresponding to the context key values specified. **get sessionServiceDetail:** Used to retrieve sessionService entity data structure corresponding to each of the sessionService key values specified in the arguments. The Proprietary XML API is implemented to provide find/add/modify/delete operations on the publisher list, i.e., authorized users of the system. We adapt semantics for the proprietary XML API from existing UDDI Specifications. This XML API is as in the following: **find publisher:** Used to find publishers registered with the system matching the conditions specified in the arguments. **get publisherDetail:** Used to retrieve detailed information regarding one or more publishers with given publisherID(s). **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. The Security XML API is used to enable authenticated access to the service. We adopt the semantics from existing UDDI Specifications. The Security API includes the following function calls. **get_authToken:** Used to request an authentication token as an 'authInfo' (authentication information) element from the service. The authInfo element allows the system implement access control. To this end, both the publication and inquiry API set include authentication information in their input arguments. **discard_ authToken:** Used to inform the hybrid service that an authentication token is no longer required and should be considered invalid.

Using WS-Context Schema XML API: Given the capabilities of the WS-Context Service, one can simply populate metadata instances using the WS-Context XML API as in the following scenario. Say, a user publishes a metadata under an already created session. In this case, the user first constructs a context entity element. Here, a context entity is used to represent interaction-dependent, dynamic metadata associated with a session or a service or both. Each context entity has both system-defined and user-defined identifiers. The uniqueness of the system-defined identifier is ensured by the system itself, whereas, the user-defined identifier is simply used to enable users to manage their memory space in the context service. As an example, we can illustrate a context as in ((system-defined-uuid, user-defined-uuid, "Job completed")). A context entity can be also associated with service entity and it has a lifetime. Contexts may be arranged in parent-child relationships. One can create a hierarchical session tree where each branch can be used as an information holder for contexts with similar characteristics. This enables the system to be queried for contexts associated to a session under consideration. This enables the system to track the associations between sessions. As the context elements are constructed, they can be published with save_context function of the WS-Context XML API. On receiving publishing metadata request, the system processes the request, extracts the context entity instance, assigns a unique identifier, stores in the in-memory storage and returns a respond back to the client.

4.3. The Glue Schema Specification

The Grid Laboratory Uniform Environment (Glue) Schema [17] is a collaboration effort to support interoperability between US and Europe Grid Projects. It presents description of core Grid resources at the conceptual level by defining an information model. The Glue Schema has the following core entities: site, computing element, storage element, service. The site entity is used to aggregate services and resources installed and managed by the same people. The computing element entity is a concept that captures information related computing resources. The storage element entity presents a data model for abstracting storage resources. The service entity captures all the common attributes associated to Grid Services. A site can aggregate one to n computing elements, one to n storage elements, one to n services. Here, each service may contain one to n service data.

In order to be compatible with the aforementioned Grid Interoperation Now (GIN) research activity and its participating Grid projects, we integrate the Glue Schema and its communication protocol with the Hybrid Service. Note that, in the prototype implementation, we showed that the proposed architecture supports the two information service implementations: Extended UDDI and WS-Context. Based on experimental study with prototype implementation and the generic architecture of the Hybrid Service, we think that existing implementations of Glue Schema Specification can easily be integrated with the proposed architecture. Thus, we do not provide an implementation for the Glue Schema. For an extensive discussion on the Glue Schema Information Model, we refer the readers to Glue Schema Specification document which is available in [17].

4.4. The Unified Schema Specification

We introduced an abstract data model and query/publish XML API for a Unified Schema Specification. We achieved the Unified Schema, which integrates the extended UDDI, the WS-Context and the Glue Schemas, by using the schema integration technique.

Schema integration is an activity of providing a unified representation of multiple data models [40]. The schema integration consists of two core steps: schema matching [40] and schema merging [41]. The schema matching step identifies mapping between the similar entities of schemas. Matching between different schema entities are defined based on semantic relationships according to the comparison of their intentional domains. To provide schema matching we have two steps: a) finding the matching concepts, b) finding the semantic relationship and constructing partial integrated schemas among the matching concepts. The schema-merging step merges different schemas and creates an integrated schema based on the mappings identified during schema matching step. The schema-merging step also identifies the mappings between the integrated schema and local schemas.

We consider the schemas ExtendedUDDI, Glue and WS-Context as a motivating example to create the Unified Schema. We start the schema integration between the ExtendedUDDI and Glue Schemas. In the first step (schema matching step), we find the following correspondences between the entities of these schemas. The first mapping is between ExtendedUDDI.businessEntity and Glue.site entities: The ExtendedUDDI. businessEntity is used to aggregate one-to-many Web Services managed by the same people or organization. Similarly, the Glue.site entity is used to aggregate services and resources managed by same people. Therefore, businessEntity and site are matching concepts, as their intentional domains are similar. The cardinality between the site and businessEntity differs, as the businessEntity may contain one-to-many site entities. For an example, Indiana University could be an instance of the businessEntity while the Community Grids Laboratory could be an instance of the site entity. Indiana University contains one-to-many research labs. The second mapping is between ExtendedUDDI.businessService and Glue.service entities: These entities are equivalent as the set of real objects that they represent are the same. The cardinality between these entities is also the same. In the integrated schema, we unify these entities as service entity. The third mapping is between ExtendedUDDI.serviceAttribute and Glue.serviceData: These two entities can be considered as equivalent as they both describe attributes associated to Grid/Web Services. The cardinality between these entities is also the same. In the integrated schema, we unify these entities as metadata. After the schema matching is completed, we merge the two schemas and create an integrated schema (ExtendedUDDI &Glue) based on the mappings that we identified.

We continue with the schema integration by integrating the WS-Context Schema with the newly constructed ExtendedUDDI&Glue Schema. In the schema-matching step, we find the following mappings: First mapping is between (ExtendedUDDI&Glue).businessEntity, (ExtendedUDDI&Glue).site and WS-Context.sessionEntity: The businessEntity is used to aggregate one-to-many services and sites managed by the same people. The site entity aggregates grid resources including services, computing and storage elements. The sessionEntity is used to aggregate session services participating to a session. Therefore, businessEntity and site (from ExtendedUDDI&Glue schema) can be considered as matching concepts with the sessionEntity (from WS-Context schema) as their intentional domains are similar. The cardinality between these entities differs, as the businessEntity may contain one to may sessionEntities. The site entity also may contain one-to-many sessionEntities. The second mapping is between: (ExtendedUDDI&Glue). service and WS-Context.sessionService: These entities are equivalent as the intentional domains that they represent are the same. The cardinality between these entities is also the same. In the integrated schema, we unify these entities as service entity. The third mapping is between (ExtendedUDDI&Glue).metadata and WS-Context.context: These entities are equivalent as the intentional domains that they represent are the same. The cardinality between these entities is also the same. In the integrated schema, we unify these entities as metadata entity. Finally, we merge the two schemas based on the mappings that we identified and create a unified schema (see Figure 9 for illustration) that integrates the Extended UDDI, WS-Context and Glue Schemas.

The Unified Schema captures both interaction-dependent and interaction-independent information associated to Grid/Web Services. The Unified Schema unifies matching and disjoint entities of different schemas.

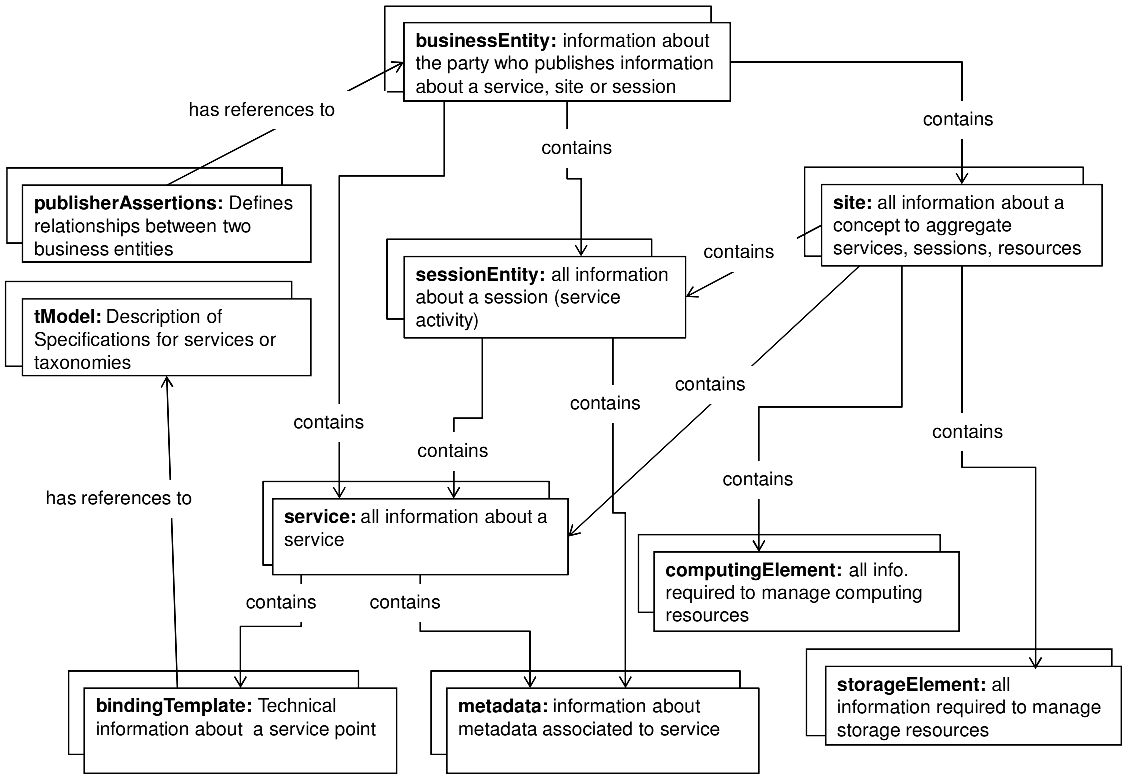

businessEntity: information about the party who publishes information about a service, site or session
has references to
contains
contains
publisherAssertions: Defines relationships between two business entities
site: all information about a concept to aggregate services, sessions, resources
sessionEntity: all information about a session (service activity)
contains
tModel: Description of Specifications for services or taxonomies
contains
contains
contains
contains
contains
contains
has references to
service: all information about a service
computingElement: all info. required to manage computing resources
contains
contains
bindingTemplate: Technical information about a service point
metadata: information about metadata associated to service
storageElement: all information required to manage storage resources

Figure 9 Unified Schema

As illustrated in Figure 9, it is comprised of the following entities: businessEntity, sessionEntity, site, service, computingElement, storageElement, bindingTemplate, metadata, tModel, publisherAssertions. A businessEntity describes a party who publishes information about a session (in other words service activity), site or service. The publisherAssertions entity defines the relationship between the two businessEntities. The sessionEntity describes information about a service activity that takes place. A sessionEntity may contain one-to-many service and metadata entities. The site entity describes information about services, their sessions and resources installed and managed by the same people. The site entity may contain information about Grid resources, such as services, computingElements and storageElements. The service entity provides descriptive information about a Grid/Web Service family. It may contain one-to-many bindingTemplate entities that define the technical information about a service end-point. A bindingTemplate entity contains references to tModel that defines descriptions of specifications for service end-points. The service entity may also have one-to-many metadata attached to it. A metadata contains information about both interaction-dependent, interaction-independent metadata and service data associated to Grid/Web Services. A metadata entity describes the information pieces associated to services or sites or sessions as (name, value) pairs.

<u>The Unified Schema XML API:</u> To facilitate testing of the federation capability, we introduce a limited Query/Publish XML API that can be carried out on the instances of the parts of the Unified Schema. We can group the Unified Schema XML API under two categories: Publish and Inquiry.

Function	Category	Information Service
Save_business	Publish	The Unified Schema XML API: This API is to support/handle both interaction-independent and interaction-dependent metadata associated to services. It enables a query/publish syntax on the heterogeneous information coming from different information service providers.
Save_session		
Save_service		
Save_metadata		
Delete_business		
Delete_session		
Delete_service		
Delete_metadata		
Get_businessDetail	Inquiry	
Get_sessionDetail		
Get_serviceDetail		
Get_metadataDetail		
Find_business		
Find_session		
Find_service		
Find_metadata		

Table 3 The Publish/Inquiry XML API for the Unified Schema. The Unified Schema XML API is introduced to enable different information service providers/clients to publish/query metadata to the Hybrid Service.

Table 3 gives the list of available XML API, which we introduce with the Unified Schema Specification. The Publish XML API is used to publish metadata instances belonging to different entities of the Unified Schema. It consists of the following functions: **save business:** Used to add/update one or more business entities into the hybrid service. **save session:** Used to add/update one or more session entities into the hybrid service. Each session may contain one-to-many metadata, one-to-many service entities and have a lifetime (lease). **save service:** Used to add/update one or more service entries into the hybrid service. Each service entity may contain one-to-many metadata element and may have a lifetime (lease). **save metadata:** Used to register or update one or more metadata associated with a service. **delete business:** Used to delete one or more business entity structures. **delete session:** Used to delete one or more sessionEntity structures. **delete service:** Used to delete one or more service entity structures. **delete metadata:** Used to delete existing metadata elements from the hybrid service. The Inquiry XML API is used to pose inquiries and to retrieve metadata from service. It consists of following functions: **find business:** This API call locates specific businesses within the hybrid services. **find session:** Used to find sessionEntity elements. The find session API call returns a session list matching the conditions specified in the arguments. **find service:** Used to locate specific services within the hybrid service. **find metadata:** Used to find service entity elements. The find service API call returns a service list matching the criteria specified in the arguments. **get businessDetail:** Used to retrieve businessEntity data structure of the Unified Schema corresponding to each of the business key values specified in the arguments. **get sessionDetail:** Used to retrieve sessionEntity data structure corresponding to each of the session key values specified in the arguments. **get serviceDetail:** Used to retrieve service entity data structure corresponding to each of the service key values specified in the arguments. **get metadataDetail:** Used to retrieve the metadata structure corresponding to the metadata key values specified.

Using the Unified Schema XML API: Given these capabilities, one can simply populate the Hybrid Service with Unified Schema metadata instances using its XML API as in the following scenario. Say, a user wants to publish both session-related and interaction-independent metadata associated to an existing service. In this case, the user constructs metadata entity instance. Each metadata entity has both system-defined and user-defined identifiers. The uniqueness of the system-defined identifier is ensured by the system itself, whereas, the user-defined identifier is simply used to enable users to manage their memory space in the context service. As an example, we can illustrate a context as in the following examples: a) ((throughput, 0.9)) and b) ((system-defined-uuid, user-defined-uuid, “Job completed”)). A metadata entity can be also associated with site, or sessionEntity of the Unified Schema and it has a lifetime. As the metadata entity instances are constructed, they can be published with "save_metadata" function of the Unified Schema XML API. On receiving publishing metadata request, the system processes the request, extracts the metadata entity instance, assigns a unique identifier, stores in the in-memory storage and returns a respond back to the client.

4.5. The Hybrid Service Semantics

The Hybrid Service introduces an abstraction layer for uniform access interface to be able to support one-to-many information service specification (such as WS-Context, Extended UDDI, or Unified Schema).

To achieve the uniform access capability, the system presents two XML Schemas: a) **Hybrid Schema** and b) **Specification Metadata (SpecMetadata) Schema**. The Hybrid Schema defines the generic access interface to the Hybrid Service. The SpecMetadata Schema defines the necessary information required by the Hybrid Service to be able to process instances of supported information service schemas. We discuss the semantics of the Hybrid Schema and the SpecMetadata Schema in the following sections.

4.5.1. The Hybrid Schema

The Hybrid Service presents an XML Schema, called the Hybrid Schema, to enable uniform access to the system. This Schema is designed to achieve a unifying access interface to the Hybrid Service. Thus, it is independent from any of the local information service schemas supported by the Hybrid Service. It defines a set of XML API to enable clients/providers to send specification-based publish/query requests (such as WS-Context's "save_context" request) in a generic way to the system.

Function	**Category**	**Information Service**
hybrid_function	Publish/Inquiry	**Hybrid Service XML API:** This API is to support unifying access interface to the Hybrid Service.
save_schemaEntity	Publish	
delete_schemaEntity		
find_schemaEntity	Inquiry	
get_schemaEntity		

Table 4 The Publish/Inquiry XML API for the Hybrid Service. The Hybrid Service XML API allows the system support one-to-many information service communication protocols.

Table 4 gives the list of available XML API, which we introduce with the Hybrid Service. It consists of the following functions: **hybrid_service:** This XML API call is used to pose inquiry/publish requests based on any specification. With this function, the user can specify the type of the schema and the function. This function allows users to access an information service back-end directly. The user also specifies the specification-based publish/query request in XML format based on the specification under consideration. On receiving the hybrid_function request call, the system handles the request based on the schema and function specified in the query. **save_schemaEntity:** This API call is used to save an instance of any schema entities of a given specification. The save_schemaEntity API call is used to update/add one or more schema entity elements into the Hybrid Grid Information Service. On receiving a save_schemaEntity publication request message, the system processes the incoming message based on information given in the mapping file of the schema under consideration. Then, the system stores the newly-inserted schema entity instances into the in-memory storage. **delete_schemaEntity:** The delete_schemaEntity is used to delete an instance of any schema entities of a given specification. The delete_schemaEntity API call deletes existing service entities associated with the specified key(s) from the system. On receiving a schema entity deletion request message, the system processes the incoming message based on information given in the mapping file of the schema under consideration. Then the system, deletes the correct entity associated with the key. **find_schemaEntity:** This API call locates schemaEntities whose entity types are identified in the arguments. This function allows the user to locate a schema entity among the heterogeneous metadata space. On receiving a find_schemaEntity request message, the system processes the incoming message based on information given in the schema mapping file of the schema under consideration. Then the system, locates the correct entities matching the query under consideration. **get_schemaEntity:** The get_schemaEntityDetail is used to retrieve an instance of any schema entities of a given specification. It returns the entity structure corresponding to key(s) specified in the query. On receiving a get_schemaEntityDetail retrieval request message, the system processes the incoming message based on information given in the mapping file of the schema under consideration. Then the system retrieves the correct entity associated with the key. Finally, the system sends the result to the user.

To illustrate the Hybrid Service access interface, we discuss the "save_schemaEntity" element (see Figure 10), which is used to publish metadata instances into the Hybrid Service.

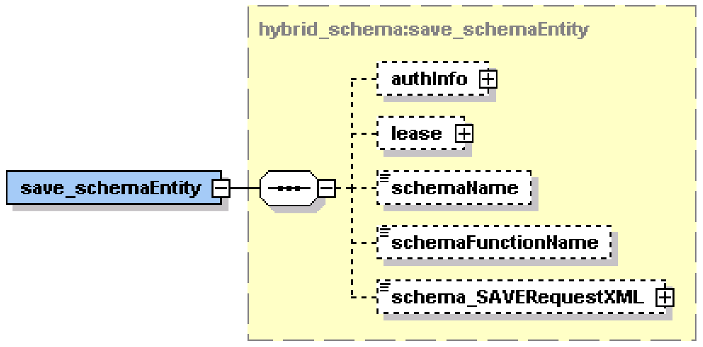


Figure 10 Hybrid Service XML Schema for Hybrid Service metadata publish function

One utilizes the "save_schemaEntity" element to publish metadata instances for the customized implementations of information service specifications. The "save_schemaEntity" element includes an "authInfo" element, which describes the authentication information; "lease" element, which is used to identify the lifetime of the metadata instance; "schemaName" element, which is used to identify a specification schema (such as Extended UDDI Schema); "schemaFunctionName", which is used to identify the function of the schema (such as "save_ serviceAttribute"); "schema_SAVERequestXML", which is an abstract element used for passing the actual XML document of the specific publish function of a given specification. The Hybrid Service requires a specification metadata document that describes all necessary information to be able to process XML API of the schema under consideration. We discuss the specification metadata semantics in the following section.

4.5.2. The SpecMetadata Schema

The SpecMetadata XML Schema is used to define all necessary information required for the Hybrid Service to support an implementation of information service specification. The structure diagram for specification metadata is illustrated in Figure 11. The Hybrid System requires an XML metadata document, which is generated based on the SpecMetadata Schema, for each information service specification supported by the system. The SpecMetadata XML file helps the Hybrid System to know how to process instances of a given specification XML API.

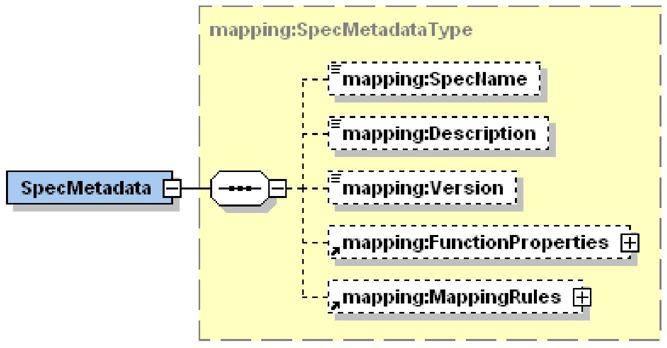


Figure 11 Structure diagram for SpecMetadata Schema: This metadata file defines all required information necessary to support a new information service

As illustrated in Figure 11, the SpecMetadata includes **Specname**, **Description**, and **Version** XML elements. These elements define descriptive information to help the Hybrid Service to identify the local information service schema under consideration. **The FunctionProperties XML element** describes all required information regarding the functions that will be supported by the Hybrid Service. The FunctionProperties element consists of one-to-many FunctionProperty sub-elements. The FunctionProperty element consists of function name, memory-mapping and information-service-backend mapping information. Here the memory-mapping information element defines all necessary information to process an incoming request for in-memory storage access. The memory-mapping information element defines the name, user-defined identifier and system-defined identifier of an entity. The information-service-backend information is needed to process the incoming request and execute the requested operation on the appropriate information service backend. This information defines the function name, its arguments, return values and the class, which needs to be executed in the information service back-end. **The MappingRules XML element** describes all required information regarding the mapping rules that provide mapping between the Unified Schema and the local information service schemas such as extended UDDI and WS-Context. The MappingRules element consists of one-to-many MappingRule sub-elements. Each MappingRule describes information about how to map a unified schema XML API to a local information service schema XML API. The MappingRule element contains the necessary information to identify functions that will be mapped to each other.

Given these capabilities, one can simply populate the Hybrid Service as in the following scenario. Say, a user wants to publish a metadata into the Hybrid Service using WS-Context's "save_context" operation through the generic access interface. In this case, firstly, the user constructs an instance of the "save_context" XML document (based on the WS-Context Specification) as if s/he wants to publish a metadata instance into the WS-Context Service. Once the specification-based publish function is constructed, it can be published into the Hybrid Service by utilizing the "save_schemaEntity" operation of the Hybrid Service Access API.

As for the arguments of the "save_schemaEntity" function, the user needs to pass the following arguments: a) authentication information, b) lifetime information, c) schemaName as "WS-Context", d) schemaFunctionName as "save_context" and e) the actual save_context document which was constructed based on the WS-Context Specification. Recall that, for each specification, the Hybrid Service requires a SpecMetadata XML document (an instance of the Specification Metadata Schema). On receipt of the "save_schemaEntity" publish operation, the Hybrid Service obtains the name of the schema (such as WS-Context) and the name of the publish operation (such as save_context) from the passing arguments. In this case, the Hybrid Service consults with the WS-Context SpecMetadata document and obtains necessary information about how to process incoming "save_context" operation. Based on the memory mapping information obtained from user-provided SpecMetadata file, the system processes the request, extracts the context metadata entity instance, assigns a unique identifier, stores in the in-memory storage and returns a response back to the client.

5 Architecture

The Hybrid Service is an add-on system that interacts with local information service and unifies them in a higher-level architecture. Figure 12 illustrates the detailed architectural design and abstraction layers of the system. The clients interact with the system through the uniform access interface. The Uniform Access layer imports the XML API of the supported Information Services. As illustrated in Figure 12, the Hybrid Information Service prototype supports XML API for Extended UDDI, WS-Context and Unified Schema (the Unified Schema integrates different local schemas into one global schema for federation of information services.). This layer is designed as generic as possible, so that it can support one-to-many XML API, as the new information services are integrated with the system. The Request-processing layer is responsible for extracting incoming requests and process operations on the Hybrid Service. It is designed to support two capabilities: notification and access control. The notification capability enables the interested clients to be notified of the state changes happening in a metadata. It is implemented by utilizing publish-subscribe based paradigm. The access control capability is responsible for enforcing controlled access to the Hybrid Grid Information Service. The investigation and implementation of access control mechanism for the decentralized information service is left out for future study. TupleSpaces Access API allows access to in-memory storage. This API supports all query/publish operations that can take place on the Tuple Pool. The Tuple Pool implements a lightweight implementation of JavaSpaces Specification [32] and is a generalized in-memory storage mechanism. It enables mutually exclusive access and associative lookup to shared data. The Tuple Processor layer is designed to process metadata stored in the Tuple Pool. Once the metadata instances are stored in the Tuple Pool as tuple objects, the system starts processing the tuples and provides the following capabilities. The first capability is the LifeTime Management. Each metadata instance may have a lifetime defined by the user. If the metadata lifetime is exceeded, then it is evicted from the TupleSpace. The second capability is the Persistency Management. The system checks with the tuple space every so often for newly added /updated tuples and stores them into the database for persistency of information. The third capability is the Fault Tolerance Management. The system checks with the tuple space every so often for newly-added/updated tuples and replicates them in other Hybrid Service instances using the publish-subscribe messaging system. This capability also provides consistency among the replicated datasets. The fourth capability is the Dynamic Caching Management. With this capability, the system keeps track of the requests coming from the pub-sub system and replicates/migrates tuples to other information services where the high demand is originated. The Filtering layer supports the federation capability. This layer provides filtering between instances of the Unified Schema and local information service schemas such as WS-Context Schema based on the user defined mapping rules to provide transformations. The Information Resource Manager layer is responsible for managing low-level information service implementations. It provides decoupling between the Hybrid Service and sub-systems. The Pub-Sub Network layer is responsible for communication between Hybrid Service instances.

5.1. Execution Logic Flow

The execution logic for the Hybrid Service happens as follows. Firstly, on receiving the client request, the request processor extracts the incoming request. The request processor processes the incoming request by checking it with the specification-mapping metadata (SpecMetadata) files. For each supported schema, there is a SpecMetadata file, which defines all the functions that can be executed on the instances of the schema under consideration. Each function defines the required information related with the schema entities to be represented in the Tuple Pool. (For example; entity name, entity identifier key, etc…). Based on this information, the request processor extracts the inquiry/publish request from the incoming message and executes these requests on the Tuple Pool. We apply the following strategy to process the incoming requests. First off all, the system keeps all locally available metadata keys in a table in the memory. On receipt of a request, the system first checks if the metadata is available in the memory by checking with the metadata-key table. If the requested metadata is not available in the local system, the request is forwarded to the Pub-Sub Manager layer to probe other Hybrid Services for the requested metadata. If the metadata is in the in-memory storage, then the request processor utilizes the Tuple Space Access API and executes the query in the Tuple Pool. In some cases, requests may require to be executed in the local information service back-end. For an example, if the client's query requires SQL query capabilities, it will be forwarded to the Information Resource Manager, which is responsible of managing the local information service implementations.

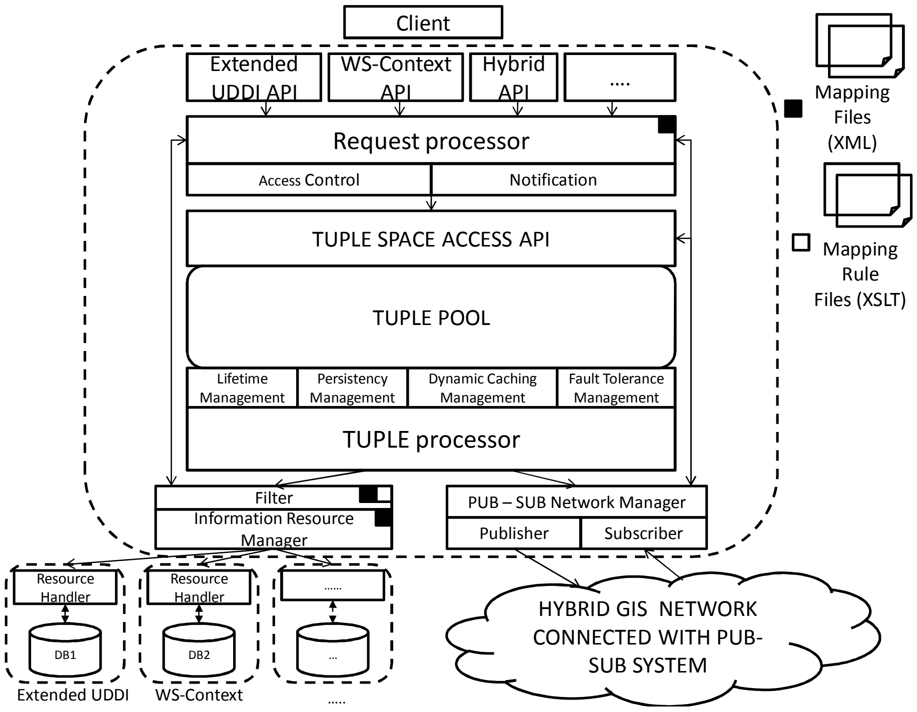


Figure 12 Execution Logic Flow for the Hybrid Grid Information Service. This figure illustrates the execution flow of the Hybrid Grid Information Service from top-to-bottom. Each rectangle shape identifies a layer of the system with particular purpose. The square-black color shapes indicate that the corresponding component checks with the specification-mapping metadata file to understand how to process the client's request. The squire-white color shape indicate that the corresponding layer checks with mapping rule files to map Unified Schema instances to appropriate local information service schema instances.

Secondly, once the request is extracted and processed, the system presents abstraction layers for some capabilities such as access control and notification. First capability is the access control management. This capability layer is intented to provide access controlling for metadata accesses. As the focus of our investigation is distributed metadata management aspects of information services, we leave out the research and implementation of this capability as future study. The second capability is the notification management. Here, the system informs the interested parties of the state changes happening in the metadata. This way the requested entities can keep track of information regarding a particular metadata instance.

Thirdly, if the request is to be handled in the memory, the Tuple Space Access API is used to enable the access to the in-memory storage. This API allows us to perform operations on the Tuple Pool. The Tuple Pool is an in-memory storage. The Tuple Pool provides a storage capability where the metadata instances of different information service schemas can be represented.

Fourthly, once the metadata instances are stored in the Tuple Pool as tuple objects, the tuple processor layer is being used to process tuples and provide a variety of capabilities. The first capability is the LifeTime Management. Each metadata instance may have a lifetime defined by the user. If the metadata lifetime is exceeded, then it is evicted from the Tuple Pool. The second capability is the Persistency Management. The system checks with the tuple space every so often for newly-added / updated tuples and stores them into the local information service back-end. The third capability is the Dynamic Caching Management. The system keeps track of the requests coming from the other Hybrid Service instances and replicates/migrates metadata to where the high demand is originated. The fourth capability is the Fault Tolerance Management. The system again checks with the tuple space every so often for newly-added / updated tuples and replicates them in other information services using the pub-sub system. This service is also responsible for providing consistency among the replicated datasets. As the main focus of this paper is to discuss information federation in Grid Information Services, the detailed discussion on replication, distribution, consistency enforcement aspects of the system is left out as the focus of another paper [7].

The Hybrid Service supports a federation capability to address the problem of providing integrated access to heterogeneous metadata. To facilitate the testing of this capability, a Unified Schema is introduced by integrating different information service schemas. If the metadata is an instance of the Unified Schema, such metadata needs to be mapped into the appropriate local information service back-end. To achieve this, the Hybrid Service utilizes the filtering layer. This layer does filtering based on the user-defined mapping rules to provide transformations between the Unified Schema instances and local schema instances. If the metadata is an instance of a local schema, then the system does not apply any filtering, and backs-up this metadata to the corresponding local information service back-end.

Fifthly, if the metadata is to be stored to the information service backend (for persistency of information), the Information Resource Management layer is used to provide connection with the back-end resource. The Information Resource Manager handles with the management of local information service implementations. It provides decoupling between the Hybrid Service and sub-systems. With the implementation of Information Resource Manager, we have provided a uniform, single interface to sub-information systems. The Resource Handler implements the sub-information system functionalities. Each information service implementation has a Resource Handler that enables interaction with the Hybrid Service.

Sixthly, if the metadata is to be replicated/stored into other Hybrid Service instances, the Pub-Sub Management Layer is used for managing interactions with the Pub-Sub network. On receiving the requests from the Tuple Processor, the Pub-Sub Manager publishes the request to the corresponding topics. The Pub-Sub Manager may also receive key-based access/storage requests from the pub-sub network. In this case, these requests will be carried out on the Tuple Pool by utilizing TupleSpace Access API. The Pub-Sub Manager utilizes publisher and subscriber sub-components in order to provide communication among the instances of the Hybrid Services.

5.2. Modular Structure

The Hybrid Grid Information Service prototype implementation consists of various modules such as Query and Publishing, Expeditor, Filter and Resource Manager, Sequencer, Access and Storage. This software is open source project and available at [42]. The Query and Publishing module is responsible for processing the incoming requests issued by end-users. The Expeditor module forms a generalized in-memory storage mechanism and provides a number of capabilities such as persistency of information. The Filter and Resource Manager modules provide decoupling between the Hybrid Information Service and the sub-systems. The Sequencer module is responsible for labeling each incoming context with a synchronized timestamp. Finally, the Access and Storage modules are responsible for actual communication between the distributed Hybrid Service nodes to support the functionalities of a replica hosting system.

The Query and Publishing module is responsible for implementing a uniform access interface for the Hybrid Grid Information Service. This module implements the Request Processing abstraction layer with access control and notification capabilities. On completing the request processing task, the Query and Publishing module utilizes the Tuple Space API to execute the request on the Tuple Pool. On completion of operation, the Query and Publication module sends the result to the client. As discussed earlier, context information may not be open to anyone, so there is a need for an information security mechanism. We leave out the investigation and implementation of this mechanism as a future study. We must note that to facilitate testing of the centralized Hybrid Service in various application use domains, we implemented a simple information security mechanism. Based on this implementation, the centralized Hybrid Service requires an authentication token to restrict who can perform inquiry/publish operation. The authorization token is obtained from the Hybrid Service at the beginning of client-server interaction. In this scenario, a client can only access 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 which needs to be passed in the argument lists of publish/inquiry operations. The Query and Publishing module also implements a notification scheme. This is achieved by utilizing a publish-subscribe based messaging scheme. This enables users of Hybrid Service to utilize a push-based information retrieval capability where the interested parties are notified of the state changes. This push-based approach reduces the server load caused by continuous information polling. We use the NaradaBrokering software [43] as the messaging infrastructure and its libraries to implement subscriber and publisher components.

The Expeditor module implements the Tuple Spaces Access API, Tuple Pool and Tuple-processing layer. The Tuple Spaces Access API provides an access interface on the Tuple Pool. The Tuple Pool is a generalized in-memory storage mechanism. Here, to meet the performance requirement of the proposed architecture, we built an in-memory storage based on the TupleSpaces paradigm [31]. The Tuple-processing layer introduces a number of capabilities: LifeTime Management, Persistency Management, Dynamic Caching Management and Fault Tolerance Management. Here, the LifeTime Manager is responsible for evicting those tuples with expired leases. The Persistency Manager is responsible for backing-up newly-stored / updated metadata into the information service back-ends. The Fault Tolerance Manager is responsible for creating replicas of the newly added metadata. The Dynamic Caching Manager is responsible for replicating/migrating metadata under high demand onto replica servers where the demand originated.

The Filtering module implements the filtering layer, which provides a mapping capability based on the user defined mapping rules. The Filtering module obtains the mapping rule information from the user-provided mapping rule files. As the mapping rule file, we use the XSL (stylesheet language for XML) Transformation (XSLT) file. The XSLT provides a general purpose XML transformation based on pre-defined mapping rules. Here, the mapping happens between the XML APIs of the Unified Schema and the local information service schemas (such as WS-Context or extended UDDI schemas).

The Information Resource Manager module, illustrated in Figure 13, handles with management of local information service implementations such as the extended UDDI. The Resource Manager module separates the Hybrid System from the sub-system classes. It knows which sub-system classes are responsible for a request and what method needs to be executed by processing the specification-mapping metadata file that belongs the local information service under consideration. On receipt of a request, the Information Resource Manager checks with the corresponding mapping file and obtain information about the specification-implementation. Such information could be about a class (which needs to be executed), it's function (which needs to be invoked), and function's input and output types, so that the Information Resource Manager can delegate the handling of incoming request to appropriate sub-system. By using this approach, the Hybrid Service can support one-to-many information services as long as the sub-system implementation classes and the specification-mapping metadata (SpecMetadata) files are provided. The Resource Handler is an external component to the Hybrid Service. It is used to interact with sub-information systems. Each specification has a Resource Handler, which allows interaction with the database. The Hybrid System classes communicate with the sub-information systems by sending requests to the Information Resource Manager, which forwards the requests to the appropriate sub-system implementation. Although the sub-system object (from the corresponding Resource Handler) performs the actual work, the Information Resource Manager seems as if it is doing the work from the perspective of the Hybrid Service inner-classes. This approach separates the Hybrid Service implementation from the local schema-specific implementations.

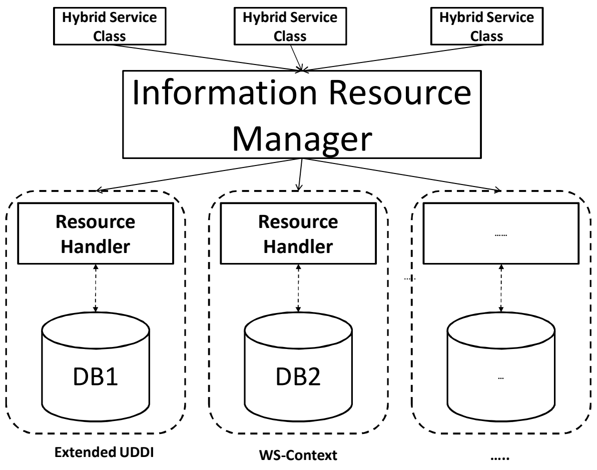


Figure 13 We implemented an Information Resource Manager, which separates specification-implementations from the implementation of the Hybrid Service.

The Resource Manager module is also used for recovery purposes. We have provided a recovery process to support persistent in-memory storage capability. This type of failure may occur if the physical memory is wiped out when power fails or machine crashes. This recovery process converts the database data to in-memory storage data (from the last backup). It runs at the bootstrap of the Hybrid Service. This process utilizes user-provided "find_schemaEntity" XML documents to retrieve instances of schema entities from the information service backend. Each "find_schemaEntity" XML document is a wrapper for schema specific "find" operations. At the bootstrap of the system, firstly, the recovery process applies the schema-specific find functions on the information service backend and retrieves metadata instances of schema entities. Secondly, the recovery process stores these metadata instances into the in-memory storage to achive persistent in-memory storage.

In order to impose an order on updates, each context has to be time-stamped before it is stored or updated in the system. The responsibility of the Sequencer module is to assign a timestamp to each metadata, which will be stored into the Hybrid Service. To do this, the Sequencer module interacts with Network Time Protocol (NTP)-based time service [44] implemented by NaradaBrokering [43] software. This service achieves synchronized timestamps by synchronizing the machine clocks with atomic timeservers available across the globe.

6 Evaluation

An evaluation study was discussed on the performance of the Hybrid Service for WS-Context XML API standard operations in [39]. It showed that the Hybrid Service provides remarkable performance achievements and addresses performance requirements of dynamic Grid/Web Service Collections. Another evaluation study that analyzes the decentralized replica hosting environment aspects of the Hybrid Service will soon be available in [7]. This study investigated the efficiency of distribution, replica-content placement and consistency enforcement aspects of the Hybrid Service and pointed out that stable, high performance, distributed Grid Information Architectures can be built by utilizing publish-subscribe based messaging schemes.

In this paper, we investigate the performance and scalability aspects of the Hybrid Service with respect to information federation. We present an evaluation of the prototype implementation of the proposed system architecture for the Unified Schema XML API standard operations. In this section, the following research questions are being addressed: What is the performance of the Hybrid Service prototype with federation capability as far as the Unified Schema XML API standard operations?, How do Unified Schema XML API functions compare against other supported Schema XML APIs such as WS-Context XML API?, What is the scalability of Hybrid Service prototype for Unified Schema XML API standard operations under increasing work load or message sizes?

The investigations are conducted 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 configuration of the cluster nodes is given at Table 5 while the software environment for the experiments is listed in Table 6. In the experiments, the performance is evaluated with respect to response time at client applications. The response time is the average time from the point a client sends off a query until the point the client receives a complete response. Note that given client/server architecture, with all machines on the same network, is setup to measure an approximation of the optimal system performance. The results measured in this environment will be the optimal upper bound of the system performance. Analyzing the results gathered from the experiments, we encountered some outliers. External effects such as network and server mainly cause these outliers, as we did not see these abnormal values in the internal timing-data, which is obtained by measuring the plain processing time. In order to avoid abnormalities in the results, we removed the outliers by utilizing the Z-filtering methodology that discards the anomalies.

Hardware configuration	
Processor	*Intel® Xeon™ CPU (2.40GHz)*
RAM	*2GB total*
Network Bandwidth	*100 Ambits/sec.*[1] *(among the cluster nodes)*
OS	*GNU/Linux (kernel release 2.4.22)*

Table 5 Summary of the cluster node - machine configurations

Software configuration	
Compiler	*Java 2 Standard Edition v.1.5 with maximum heap size of 1024 MB using the –Xmx1024m option*
Servlet container	*Tomcat Apache Server v.5.5.8 with max. multiple thread number of 1000*
Web Service container	*Apache Axis v.2.0*
Database	*MYSQL with v.4.1*
Timing function	*Java 2 with v.1.5 – timing function "nanoTime()"*

Table 6 Software environment configuration

[1] The bandwidth measurements were taken with Iperf tool for measuring TCP and UDP bandwidth performance.(http://dast.nlanr.net/Projects/Iperf)

The performance of the system is tested with a multithreaded client program that takes the following arguments: a) the number of threads (N) and b) number of messages (T) to be fired by each thread. We illustrate our timing methodology in the pseudo code below.

```
SET the number of threads to N
SET the number of transaction to be executed to T
CREATE N number of threats
STOP the threads until N threads are created and ready
ThreadSleep(random(1000))
FOR X = 1 to T
    SET start to 0, stop to 0
    START time

    Hybrid_Service_API(..)

    STOP time
    PRINT  (elapse time)
END FOR
```

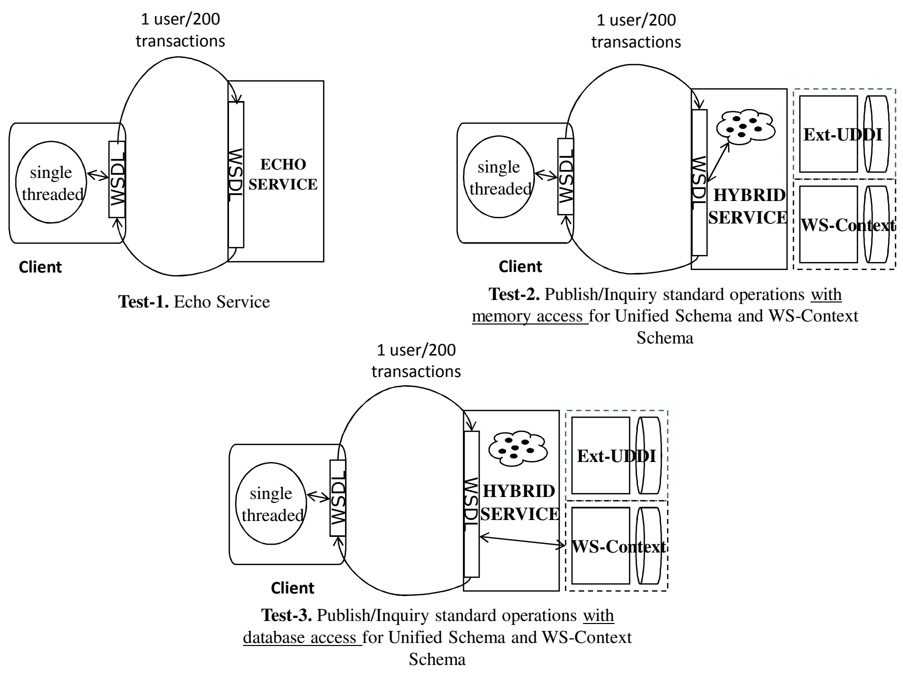
We conducted two experiments to understand the behavior of the system with respect to information federation. These are **performance** and **scalability** experiments.

The **performance experiment** is conducted to understand the baseline performance of the prototype implementation of the Hybrid Service. This evaluation investigates the performance of system for standard Unified Schema operations and compares it against the performance of WS-Context Shema operations when there is no additional traffic. To do this following testing cases are completed: a single client sends publish requests to an echo service which receives a message and then sends it back to the client with no processing applied; a single client sends publish requests to a Hybrid Service which grants the request with memory access; a single client sends publish requests to a Hybrid Service which grants the request with database access. In the experiment, both the Hybrid Service and testing client application were located in two different servers located in the Linux cluster. The design of these experiment is depicted in Figure 14 and simulation parameters are given in Table 7. This experiment was repeated five times and we record the average response time.

In our earlier performance evaluation study on the Hybrid Service [39], we investigated the best possible backup-interval period to provide persistency at high performance response rate. In this investigation, we observed 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 publish function is ~ 10.2 milliseconds. If the backup frequency is every 10 seconds or lower, we find that average execution time for publish operation stabilized to ~7.5 milliseconds. Therefore, we choose the value for backup frequency as every 10 sec in the experiments. Here, for testing purposes, we used WS-Context Schema primary operations: save_context and get_context and the equivalent Unified Schema primary operations: save_metadata and get_metadata.

Simulation parameters	
Metadata size	*1.7 KB*
Registry size	*5000 metadata*
Backup interval time	*Every 10 milliseconds*
Observation number	*200*

Table 7 Simulation parameters. Some metadata examples for the experiments are given in the appendix.



Test-1. Echo Service

Test-2. Publish/Inquiry standard operations with memory access for Unified Schema and WS-Context Schema

Test-3. Publish/Inquiry standard operations with database access for Unified Schema and WS-Context Schema

Figure 14 Testing cases of responsiveness experiment for Unified Schema and WS-Context standard operations

Analyzing the results depicted in Figure 15 and Figure 16 and listed in Table 8 and Table 9, we observe that the Hybrid Service has negligible processing overheads when the federation capability is being used. This experimental study indicates that the Hybrid Service achieves noticeable performance improvements in metadata management for standard operations by simply employing an in-memory storage mechanism, while preserving a certain persistency level. We also observe that the Unified Schema operations require more time (as opposed to WS-Context Schema operations) for database accesses. This is because the system keeps the Unified Schema metadata in the relevant local information service (in this case WS-Context XML Metadata Service) for persistency reasons. In turn, the system requires additional time for database accesses to perform transformation between the Unified Schema and WS-Context Schema instances.

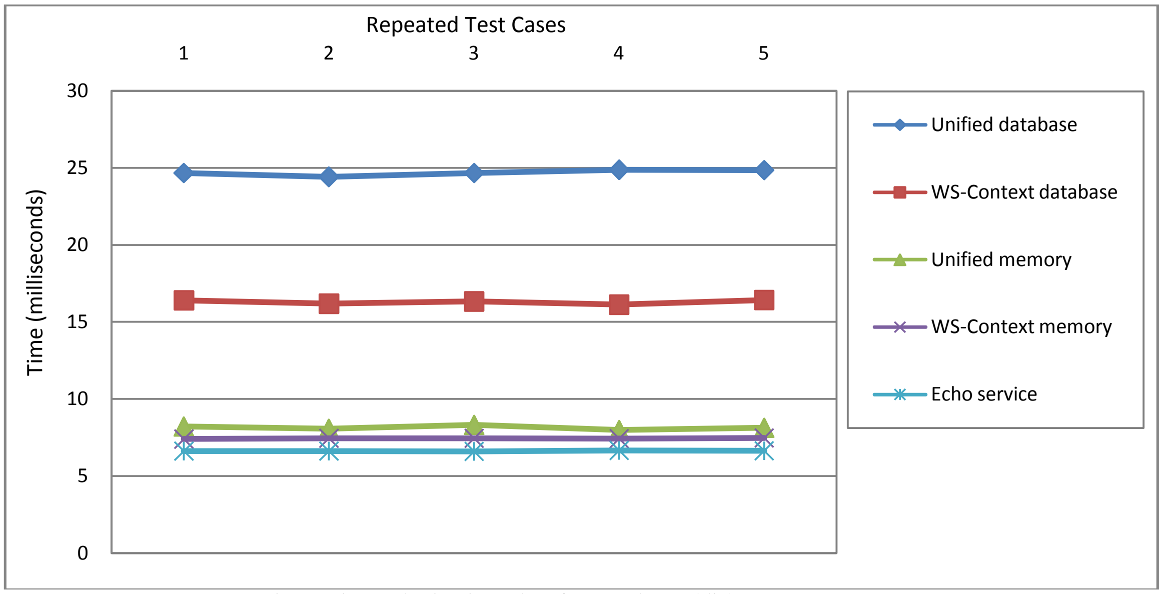


Figure 15 Round Trip Time Chart for Metadata Publish Requests

Statistics for different publish request testing cases		
	Average timings	STDev
Test-1 – Echo Service	6.64	1.40
Test-2 – Unified - memory	8.08	1.13
Test-3 – WSContext - memory	7.46	1.40
Test-4 – Unified - database	24.41	1.78
Test-5 – WS-Context - database	16.19	2.06

Table 8 Statistics for the performance experiment. We conduct testing cases to learn performance of the Unified Schema standard publish operations. Test-1: Echo service testing case, Test-2: Unified Schema publish-operation with memory access testing case, Test-3: WS-Context Schema publish-operation with memory access testing case, Test-4: Unified Schema publish-operation with database access testing case, Test-5: WS-Context Schema publish-operation with database access testing case. The time units are in milliseconds.

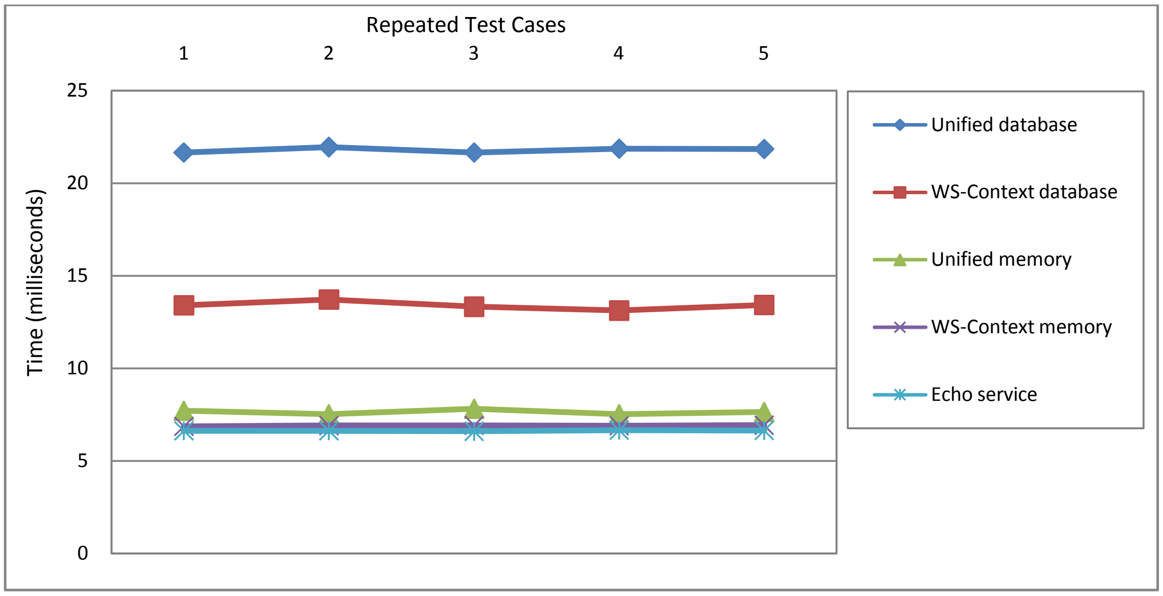


Figure 16 Round Trip Time Chart for Metadata Inquiry Requests

Statistics for different inquiry request testing cases		
	Average timings	STDev
Test-1 – Echo Service	6.64	1.40
Test-2 – Unified - memory	7.54	1.36
Test-3 – WS-Context - memory	6.92	1.70
Test-4 – Unified - database	21.95	1.67
Test-5 – WS-Context - database	13.73	2.08

Table 9 Statistics for the performance experiment. We conduct testing cases to learn performance of the Unified Schema standard publish operations. Test-1: Echo service testing case, Test-2: Unified Schema publish-operation with memory access testing case, Test-3: WS-Context Schema publish-operation with memory access testing case, Test-4: Unified Schema publish-operation with database access testing case, Test-5: WS-Context Schema publish-operation with database access testing case. The time units are in milliseconds.

In the **scalibility experiment,** we investigated two research questions: a) how well does the Hybrid Service perform when the context size is increased; b) how well does the Hybrid Service perform when the message rate per second is increased. In this experiment we investigated the performance of the Unified Schema XML API to understand the system behavior under increasing workloads while the federation capability is being used.

To answer the first research question, as illustrated in Test-A in Figure 17, we increased the context sizes at each step of the experiment, until we observe the degradation in the response times. To answer the second question, as illustrated in Test-B in Figure 17, we ramped-up the work load (number of messages sent per second) until the system performance degrades.

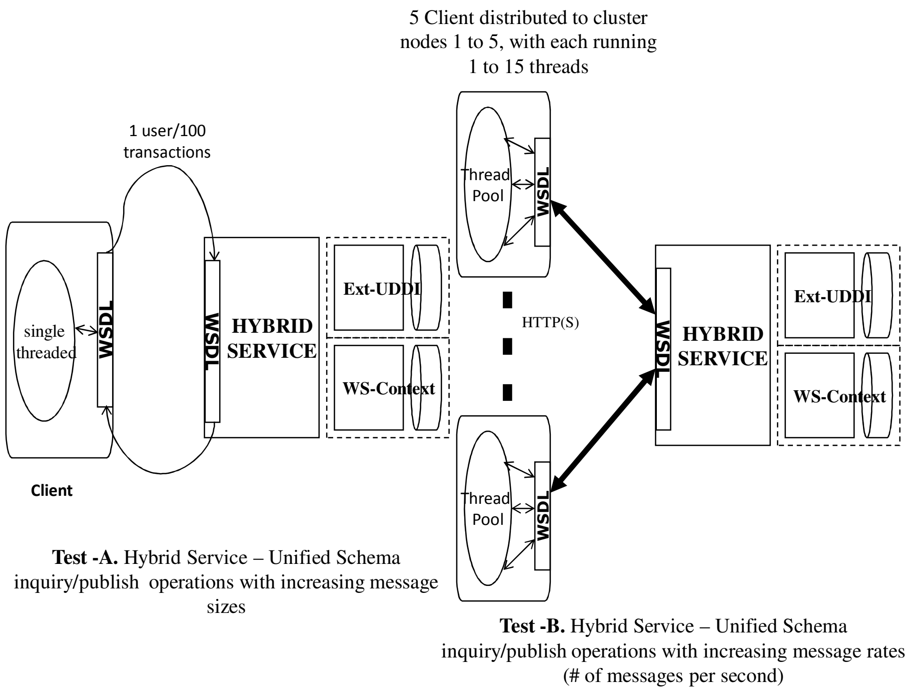

5 Client distributed to cluster nodes 1 to 5, with each running 1 to 15 threads
1 user/100 transactions
single threaded
WSDL
Client
WSDL
HYBRID SERVICE
Ext-UDDI
WS-Context
Thread Pool
WSDL
HTTP(S)
Thread Pool
WSDL
WSDL
HYBRID SERVICE
Ext-UDDI
WS-Context
Test -A. Hybrid Service – Unified Schema inquiry/publish operations with increasing message sizes
Test -B. Hybrid Service – Unified Schema inquiry/publish operations with increasing message rates (# of messages per second)

Figure 17 Testing cases of scalibility experiment for Unified Schema inquiry and publish functionalities

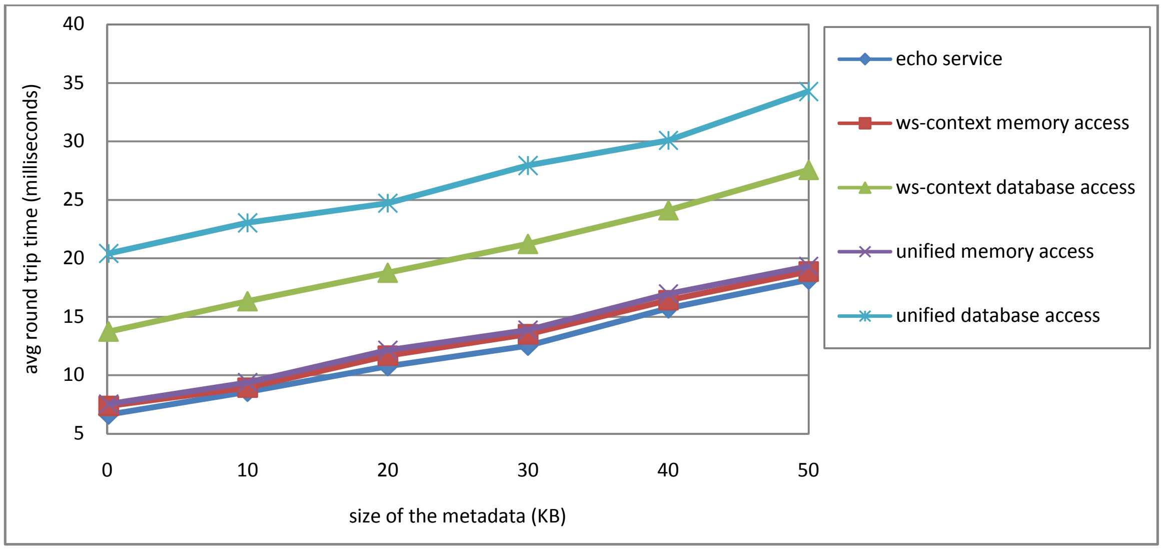


Figure 18 Round Trip Time chart for publish requests for increasing metadata payload sizes

	echo service		ws-context memory access		ws-context database access		unified-schema memory access		unified-schema database access	
KB	Avg	STDev	Avg	STDev	Avg	STDev	Avg	STDev	Avg	STDev
0.1	6.63	1.55	7.38	1.70	13.72	1.59	7.94	1.69	20.42	1.66
10	8.58	1.67	8.93	1.67	16.33	1.79	9.37	1.49	23.03	1.63
20	10.78	1.66	11.68	1.67	18.78	1.86	12.14	1.77	24.71	1.72
30	12.52	1.72	13.50	1.74	21.23	1.76	13.87	1.66	27.94	1.64
40	15.72	1.67	16.42	1.67	24.12	1.62	16.95	1.72	30.08	1.52
50	18.17	1.73	18.87	1.75	27.57	1.65	19.33	1.55	34.27	1.59

Table 10 Statistics of Figure 18 for Hybrid Service – Unified Schema - publish operations with changing context payload sizes. Time units are in milliseconds.

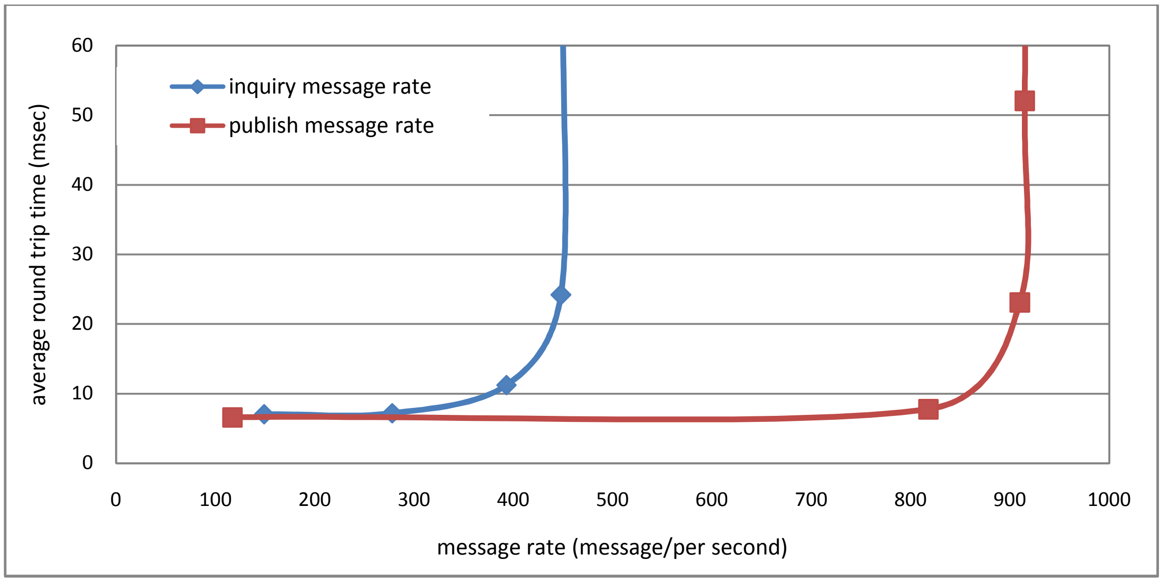


Figure 19 Unified Schema inquiry and publish response time at various levels of message rates per second

Hybrid Service – Unified Schema inquiry operation		
messages/second	average timings	STDev
117	6.58	0.73
818	7.74	0.96
910	23.08	15.42
915	52.05	27.34
916	107.12	54.73
Hybrid Service – Unified Schema publish operation		
messages/second	average timings	STDev
149	7.04	1.85
278	7.17	1.97
393	11.22	6.65
448	24.17	13.18
450	64.73	38.97

Table 11 Statistics of the experiment results depicted in Figure 19. These measurements were taken with Hybrid Service when the Unified Schema inquiry and publish request is granted with memory access. Time units are in milliseconds.

The results of this experiment are depicted in Figure 18 and Figure 19 and listed in Table 10 and Table 11. Analyzing the results depicted in Figure 18, we concluded that Hybrid Service Unified Schema XML API standard operations performed well for increasing message sizes. By comparing the performance values from an Echo Service and Hybrid Service, we observe that pure server processing time is negligible and remains the same as the size of the messages increases. Analyzing the results depicted in Figure 19, we conclude that Hybrid Service Unified Schema XML API standard operations performed well under increasing message rates. For inquiry request messages, we observe a threshold value after which the system performance starts decreasing due to high message rate. 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. For publish request messages, we observe another threshold value where the system performance starts dropping down. The reason for this is the following. As the publish message-rate is increased, the number of updated/newly written metadata in the Tuple Pool is also increased. In turn, the action that writes and transformes the larger number of updates into the default local information service back-end affects the system performance and causes higher fluctuations in the response times for increasing number of simultaneous publish requests.

7 Conclusion and Future Research Directions

We introduced a novel architecture for a Hybrid Grid Information Service (Hybrid Service) supporting handling and discovery of not only quasi-static, stateless metadata, but also session related metadata. The Hybrid Service is an add-on architecture that runs one layer above existing information service implementations. It provides unification, federation and interoperability of Grid Information Services. To achieve unification, the Hybrid Service is designed as a generic system with front and back-end abstraction layers supporting one-to-many local information systems and their communication protocols. To achieve federation, the Hybrid Service is designed to support information integration technique in which metadata from several heterogeneous sources are transferred into a global schema and queried with a uniform query interface. To manage both quasi-static and dynamic metadata and provide interoperability with wide-range of Web Service applications, the Hybrid Service is integrated with two local information services: WS-Context XML Metadata Service and Extended UDDI XML Metadata Service. The WS-Context Service is implemented based on WS-Context Specification to manage dynamic, session related metadata. It is an implementation of the Context Manager component of the WS-Context Specification. The Extended UDDI Service is implemented based on an extended version of the UDDI Specification to manage semi-static, stateless metadata.

We performed a set of experiments to evaluate the performance and scalability of the Hybrid Service to understand whether it can achieve information federation with acceptable costs. This evaluation study pointed out the following results. Firstly, it pointed that the Hybrid Service achieves information federation with negligible processing overheads for accessing/storing metadata. Secondly, it indicated that the Hybrid Service achieves noticeable performance improvements in standard operations by employing an in-memory storage while preserving persistency of information. Thirdly, it pointed that the Hybrid Service scales to high message rates and message sizes while supporting information integration where metadata comes from heterogeneous data-systems.

With this research, we revisited distributed data management techniques to achieve integrated access to heterogeneous metadata coming from limited number of local information services. We intend to further improve this approach to be able to scale up to high number of metadata sources. An additional area that we intend to research is an information security mechanism for the distributed Hybrid Service.

Acknowledgement: The Advanced Information Systems Technology Program of NASA's Earth-Sun System Technology Office supported this research.

Appendix: XML Metadata Examples

A. Context XML metadata

```
<?xml version="1.0" encoding="UTF-8"?>
<wscontext:context
        xmlns:wscontext="http://datatype.fthpis.cgl/"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <contextKey>ABCCE800-AB35-11DA-A4FC-C80C5880CB18</contextKey>
        <serviceKey>ABCCE800-AB35-11DA-A4FC-C80C5880CB19</serviceKey>
        <sessionKey>ABCCE800-AB35-11DA-A4FC-C80C5880CB20</sessionKey>
        <name>context://GIS/PI/ABCCE544-CX35-11EA-BVFC-C34C7789CB33</name>
        <value>context:///GIS/VC/3ea29661-2d5e-11db-8c56-cf37cd202027/3ebd7162-2d5e-11db-8c56-
cf37cd202027/cost</value>
        <valueType>String</valueType>
        <lease>
                <timeout>1000</timeout>
                <isInfinite>false</isInfinite>
        </lease>
        <version>1</version>
</wscontext:context>
```

B. Unified Schema XML metadata

```
<?xml version="1.0" encoding="UTF-8"?>
<unified_schema:service
        xmlns:hybrid_schema="http://datatype.generic.fthpis.cgl/"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <serviceKey>856679F0-B4B6-11DA-A1DD-E719F6E12358</serviceKey>
        <serviceType>Web Feature Service</serviceType>
        <name>Service Name</name>
        <description>
                <value>Service Description</value>
        </description>
        <serviceEndpointAddress>http://gf7.ucs.indiana.edu:8092/wfs-streaming-service/services/wfs</serviceEndpointAddress>
        <metadata>
                <metadataKey>7115B940-A95E-11DA-B940-CB4E3E38D98F</metadataKey>
                <serviceKey>856679F0-B4B6-11DA-A1DD-E719F6E12358</serviceKey>
                <name>session-id</name>
                <value>0001</value>
                <lease><isInfinite>true</isInfinite></lease>
                <version>1</version>
        </metadata>
        <lease><isInfinite>true</isInfinite></lease>
</unified_schema:service>
```

Bibliography

1. Aktas, M.S., et al., iSERVO: Implementing the International Solid Earth Research Virtual Observatory by Integrating Computational Grid and Geographical Information Web Services. PAGEOPH, 2004.
2. Wu, W., et al., Grid Service Architecture for Videoconferencing, in "Grid Computational Methods" Edited by M.P. Bekakos, G.A. Gravvanis and H.R. Arabnia.
3. Zanikolas, S., Sakellariou, R., A Taxonomy of Grid Monitoring Systems. . Future Generation Computer Systems, 21(1), 2005: p. pp. 163--188.
4. OGF Grid Interoperation Now Community Group (GIN - CG) - Web site is available at https://forge.gridforum.org/projects/gin.
5. Bunting, B., Chapman, M., Hurley, O., Little M,, Mischinkinky, J., Newcomer, E., Webber, J., and Swenson, K. , Web Services Context (WS-Context) ver 1.0 http://www.arjuna.com/library/specs/ws_caf_1-0/WS-CTX.pdf. 2003.
6. Bellwood, T., Clement, L., and von Riegen, C., UDDI Version 3.0.1: UDDI Spec Technical Committee Specification http://uddi.org/pubs/uddi-v3.0.1-20031014.htm. 2003.
7. Aktas, M.S., Fox, Geoffrey C., Pierce, Marlon E., Distributed High Performance Grid Information Service. Submitted ot Journal of Systems and Software, 2008.
8. Lenzerini, M., Data Integration: A Theoretical Perspective, in PODS: 243-246. 2002.
9. Ziegler, P., Dittrich, K., Three Decades of Data Integration - All Problems Solved?, in WCC: 3-12. 2004.
10. T. Ozsu, P.V., Principles of Distributed Database Systems. 2nd Edition, Prentice Hall, 1999.
11. Valduriez, P., Pacitti, E., Data Management in Large-scale P2P Systems. Int. Conf. on High Performance Computing for Computational Science (VecPar'2004) - LNCS 3402, Springer, 2004: p. 109-122.
12. Florescu, D., Levy, A., Mendelzon, A., Database Techniques for the World-Wide Web:A Survey. SIGMOD Record, 27(3):56-74, 1998.
13. Open Grid Forum Web Page is available at http://www.ogf.org Access date: September 2007.
14. The Enabling Grids for E-science (EGEE) project - Web site is available at http://www.eu-egee.org/ Access date: October, 2007.
15. The National Grid Service (NGS) - Web site available is at http://www.grid-support.ac.uk/, Access date: October, 2007.
16. NorduGrid project. Web site is available at http://www.nordugrid.org/, Access date: October, 2007.
17. GLUE Schema Collaboration. The GLUE Schema homepage. http://infnforge.cnaf.infn.it/glueinfomodel/.
18. OGC, The Open Geospatial Consortiom (OGC), web site available at http://www.opengis.org.
19. Open_GIS_Consortium_Inc., OWS1.2 UDDI Experiment. OpenGIS Interoperability Program Report OGC 03-028 available at http://www.opengeospatial.org/docs/03-028.pdf. 2003.
20. Sycline, Sycline Inc., web site available at http://www.synclineinc.com.
21. Galdos, Galdos Inc., web site available at http://www.galdosinc.com.

22. Dialani, V., UDDI-M Version 1.0 API Specification. 2002, University of Southampton – UK. 02.: Southampton.
23. ShaikhAli, A., Rana, O., Al-Ali, R., Walker, D. UDDIe: An Extended Registry for Web Services. Proceedings of the Service Oriented Computing: Models, Architectures and Applications. in SAINT-2003 IEEE Computer Society Press. . 2003. Orlando Florida, USA.
24. Verma, K., Sivashanmugam, K. , Sheth, A., Patil, A., Oundhakar, S. and Miller, J., METEOR–S WSDI: A Scalable P2P Infrastructure of Registries for Semantic Publication and Discovery of Web Services. Journal of Information Technology and Management.
25. GRIMOIRES - UDDI compliant Web Service registry with metadata annotation extension, availble at http://sourceforge.net/projects/grimoires.
26. MyGrid - UK e-Science project, available at http://www.mygrid.org.uk.
27. Tanenbaum, A., Van Steen, M., Distributed Systems Principles and Paradigms. 2002. Cited in page 326.
28. Fang, L., Gannon, D., XPOLA - an Extensible Capability-based Authorization Infrastructure for Grids, in In 4th Annual PKI R&D Workshop, April 2005.
29. Saltzer, J., and Schroeder, M., The protection of information in computer systems. Proceedings of the IEEE, vol.63, no. 9, pp. 1278-1308, 1975.
30. Sandhu, R.S., Coyne, E. J., Feinstein, H.L. and Youman, C. E., Role-Based Access Control Models. IEEE Computer, vol. 29, no. 2, pp. 38-47, 1996.
31. Carriero, N., Gelernter, D., Linda in Context. Commun. ACM, 32(4): 444-458, 1989.
32. Sun_Microsystems, JavaSpaces Specification Revision 1.0, 1999 available at http://www.sun.com/jini/specs/js.ps.
33. Wyckoff, P., Lehman, T. J., McLaughry, S., T Spaces. IBM Systems Journal, 1998. 37(3): p. 454-474.
34. Khushraj, D., Lassila, O., Finin, T. sTuples:Semantic Tuple Spaces. in IEEE Proceedings of the First Annual International Conference on Mobile and Ubiquitous Systems:Networking and Services (MobiQuitous'04). 2004.
35. Krummenacher, R., Strang, T., Fensel, D. Triple Spaces for and Ubiquitous Web of Services. in W3C Workshop on the Ubiquitous Web. March 2005. Tokyo, Japan.
36. Tolksdorf, R., Nixon, L., Liebsch, F., Nguyen, M.D., Bontas, P.E., Semantic Web Spaces, in Technical Report B-04-11. July 2004, Freie Univesitat Berlin, Institut fur Informatik: Berlin, Germany.
37. Coleman, r., Bhardwaj, A., Dellucca, A., Finke, G., Sofia, A., Jutt, M., Batra, S., MicroSpaces software with version 1.5.2 available at http://microspaces.sourceforge.net/. 2004.
38. Aktas, M.S., Oh, Sangyoon, Fox, Geoffrey C., Pierce, Marlon E. XML Metadata Services. in The 2nd International Conference on Semantics, Knowledge and Grid (SKG2006). 2006. Guilin, China.
39. Mehmet S. Aktas, G.C.F., Marlon Pierce, XML Metadata Services. Concurr. Comput. : Pract. Exper., 2008.
40. Rahm, E., Bernstein, P., A survey of approaches to automatic schema matching. . VLDB Journal10 (2001) 334-350.
41. Bernstein, P., Applying model management to classical meta data problems In Proc. CIDR (2003) 209-220.

42. Aktas, M.S., Fault Tolerant High Performance Information Service - FTHPIS - Hybrid WS-Context Service web site, available at http://www.opengrids.org/wscontext.
43. Pallickara, S. and G. Fox. NaradaBrokering: A Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids. in Lecture Notes in Computer Science. 2003: Springer-Verlag.
44. Bulut, H., S. Pallickara, and G. Fox. Implementing a NTP-Based Time Service within a Distributed Brokering System. in ACM International Conference on the Principles and Practice of Programming in Java, June 16-18, 2004 Las Vegas, NV. 2004.