Information Federation in Grid Information Services

MEHMET S. AKTAS

Submitted to the faculty of the Indiana University Graduate School
in partial fulfillment of requirements
for the degree
Doctoral of Philosophy
in the Department of Computer Science
Indiana University

May 2007

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Doctoral Committee

Geoffrey C. Fox, Ph.D. (Principal Advisor)

Dennis Gannon, Ph.D.

David Leake, Ph.D.

Beth Plale, Ph.D.

May 3, 2007

© 2007

Mehmet S. Aktas

All Rights Reserved

Abstract

Information Services address the challenging problems of announcing and discovering resources in Grids. 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 set of requirements: Large-scale Grid applications require management of large amounts of relatively slowly varying metadata. E-Science Grid applications such as dynamic Grid/Web Service collections require greater support for dynamic metadata. We research Grid Information Services that support both the scalability of large amounts of relatively slowly varying metadata and the performance demands of rapidly updated information in dynamic regions.

We propose a novel system architecture that provides unification and federation of information in Grid Information Services. The proposed system utilizes publish-subscribe paradigm and associative shared memory platforms to provide an add-on architecture that interacts with existing information systems. We present an empirical evaluation of our approach and investigate its practical usefulness. The results demonstrate that the proposed system improves the quality of information services in terms of performance and fault-tolerance with negligible processing overheads. The results also indicate that efficient decentralized Grid Information Service Architectures can be built by utilizing publish-subscribe based messaging schemes.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

Information Services address the challenging problems of announcing and discovering resources in Grids. 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 can 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.

Extensive metadata requirements of both the worldwide Grid and the dynamic Grid/Web Service collections that support local dynamic action may be investigated in diverse sets of application domains such as sensor and collaboration grids. For example, workflow-style Geographical Information System Grids such as the Pattern Informatics application [1] require information systems for storing both semi-static, stateless and transitory metadata needed to describe distributed session state information. The Pattern Informatics application is an earthquake simulation and modeling code integrated with streaming data services as well as visualization services for earthquake forecasting. Another example, collaborative streaming systems such as Global Multimedia Collaboration System (GlobalMMCS) [3] involve both large, mostly static information systems as well as much smaller and dynamic ones. GlobalMMCS is a service-oriented collaboration system, which integrates various services including videoconferencing, instant messaging and streaming, and is interoperable with multiple videoconferencing technologies (see Section 1.3.2 for more detailed discussion on application use domains).

Figure 1 illustrates a model of building a system hierarchy where services are aggregated into atomic grids that perform basic functionality. We assume that dynamic Grid/Web Service collections model the desired functionality. Our goal is to define the practical extent of a given dynamic Grid/Web Service collection based on information exchange. The basic (atomic) grids include Geographical Information System, collaboration, sensor, compute, and knowledge grid. Composite grids are built recursively from both atomic and other composite grids. In this picture, we need the core Grid Services at the bottom of the figure with services like XML metadata services (in other words Grid Information Services) for static and dynamic information.

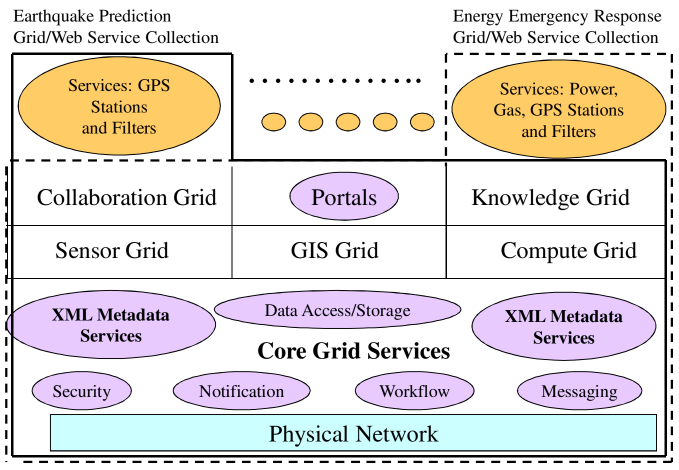


Figure 1. A dynamic Grid/Web Service collection may be built in a dynamic fashion as Grids of Grids applications with modest number of services involved at any one time for particular functionality

The basic grids can be reused in all critical infrastructure grids, which in turn are customized, compared and overlaid with other grids for different critical infrastructure communities such as crisis grid, emergency response and so forth. As an example, a Pattern Informatics application can be built in composite fashion from basic grids, such as Geographical Information System and sensor grids. Given this picture, we expect that Grid of Grids concept [4] can be applied recursively to build dynamic Grid/Web Service collections.

1.1 Motivation

As the Service Oriented Architecture (SOA) principles have gained importance, an emerging need has appeared for methodologies to locate desired services that provide access to their capability descriptions. As these services interact with each other within a workflow session to produce a common functionality, another emerging need has also appeared for storing, querying, and sharing the resulting metadata needed to describe session state information. The Grid Information Services support both discovery and handling of services through metadata and are vital components of Grids [5]. In this thesis, we are particularly interested in investigating Grid Information Services that are able to manage both stateless and stateful (transient) metadata associated to services in Service Oriented Architectures.

We identify the following limitations of current approaches in Information Services supporting Grids.

First, different Grid applications adopted customized implementations of Grid Information Services. Their data model and communication language is different. Therefore, these information services are not interoperable. They cannot share each other's metadata and utilize each other's resources.

Second, most of the existing Grid Information Services do not support dynamically assembled service collections gathered at any one time to solve a particular problem at hand [6, 7]. The main reason for this is that they are not built along this model. For instance, they do not provide capabilities (such as lifetime management, notification mechanisms, and so forth) to support dynamic metadata management. The majority of existing approaches (for an example, the Universal Description, Discovery, and Integration (UDDI) [8]) are used to discover/handle quasi-static, rarely changing information while ignoring the dynamically generated session state information.

Third, most of the existing approaches to Information Services have centralized components and do not address high performance and fault-tolerance issues [6, 7].

Handling information requirements of dynamic Grid/Web Service collections requires high performance, decentralized, and fault tolerant information systems.

Fourth, existing Information Service mechanisms do not take into account demand changes when making decisions on metadata access and storage. However, information services for dynamic regions should be able to relocate metadata to nearby locations of interested entities in order to provide efficient access, storage of the shared information, as the dynamic metadata needs to be delivered on tight time constraints within a given dynamic Grid/Web Service collection.

Fifth, existing approaches to Information Services do not provide uniform interfaces for publishing and discovery of both dynamically generated and static information. This creates a limitation on the client-end, as the users have to interact with more than one metadata service. This increases the complexity of clients and creates fat clients. We therefore see this as an important area of investigation.

1.2 Statement of research problems

In this thesis, we mainly focus on investigating a novel approach of building high performance, fault tolerant Hybrid Grid Information Service. In order to build such architecture, we particularly identify the following research questions.

- Can we implement a hybrid system architecture that unifies custom implementations of Grid Information Services to provide a common access interface to different kinds of service-metadata (such as interaction-dependent and interaction-independent) in Service Oriented Architectures?

- How can we provide federation of information among the Grid Information Services, so that they can share/exchange metadata with each other? What is a common data model and communication protocol for such federation capability?
- What is the efficient metadata access/storage strategy for such a hybrid system architecture that could speed up performance of existing Grid Information Services and that could provide persistency of information?
- What are the efficient request distribution, replica-content creation, and consistency enforcement strategies to achieve decentralized hybrid information system architecture? Can we implement these fundamental features of a decentralized system with publish-subscribe based messaging schemes? How does the system behavior change for continuous operation?
- How can we achieve a self-adopting decentralized information service architecture that can answer instantaneous client-demand changes?
- Can we support communication among Grid/Web Services with efficient mediator information service methodologies?

1.3 Scope of Research

In order to define the scope of the proposed research, we outline various requirements of the desired system architecture and its application usage scenarios.

1.3.1 Requirements of the architecture

We are interested in investigating a novel architecture for information services in order to meet the following requirements of the research problem at hand.

Uniformity: The types of information may vary in both traditional and Semantic Grids. This requires a Hybrid Grid Information Service providing a uniform interface to different kinds of metadata. Thus, the Hybrid Grid Information Service architecture should be able to unify different information systems under one unified architecture and present a common access interface.

Federation: Different Grid applications adopt customized implementations of information services. These Grid applications should be able to communicate through the Grid Information Services and utilize each other's resources. This requires information federation capability in the Hybrid Grid Information Service architecture.

Interoperability: Information should be accessible by diverse set of consumer services through standard interfaces to increase usability. This requires leveraging existing Web Service standards for service discovery and communication to enable Information Services and consumer services to operate effectively together.

Dynamism: Dynamic metadata may have changing user demands over time. Therefore, metadata need to be reallocated based on changing user demands and locations. This requires Information Services that can support optimization techniques in metadata access and that can move the highly requested metadata to where they wanted.

Fault-tolerance: The Hybrid Grid Information Service architecture is required to improve the capabilities of existing information services in terms of fault tolerance. Archiving of metadata should be provided for persistency of information. Furthermore, high availability of information is necessary to keep redundant copies of the same data for fault-tolerance reasons.

Performance: The Hybrid Grid Information Service architecture is required to improve the capabilities of existing information services in terms of performance. For example, the system should be able to support dynamic, high-frequency metadata generation in a dynamic Grid/Web Service collection with a fine-granularity time delay.

1.3.2 Application use domains

To present the applicability of proposed research, we investigate the metadata management components and information requirements of various application use domains. The first example is a workflow-session metadata manager, a vital component of workflow-style Grid applications. A workflow-session metadata manager is responsible for providing a store/access/search interface to metadata generated during workflow execution. The second example is a metadata catalog service. A catalog service is a metadata service that stores both prescriptive and descriptive information about Grid/Web Services. The third example is a metadata archival service. This service is used for supporting distributed and collaborative computational science applications where user inputs needed to be archived within a session to enable users to access/reuse their previously stored user-system interactions. The fourth example is a third-party metadata repository, also called a Context-store. This component is used in a fast web service communication model in a collaborative mobile computing environment where the redundant parts of the exchanged messages are stored. Finally, the session metadata manager component is designed to provide storage/access/search interface to dynamic metadata generated in real-time conferencing applications.

1.3.2.1 A workflow session metadata manager component

Description: A workflow-session metadata manager is responsible for storing transient metadata needed to describe distributed session state information in a workflow.

Requirements: Participants of a workflow must know about the state of the system, so that they can perform their assigned tasks within a specific sequence. This can be done by either pull or push based approaches. In a pull-based approach, each participant continuously checks with the system if the state is changed. For instance, some application domains may employ various browser-based applications and pushing the states to the web-applications through an http server is rather complicated. So, the pull-based approach can be used in those domains to interact with the service to get the state updates. In a push-based approach, participants are notified of the state changes. The push-based approach is mainly used to interact with the workflow session metadata manager in order to reduce the server load caused by continuous information polling.

Usage Scenario: We have investigated two practical example usage domains, which are in need of a workflow-session metadata manager: Pattern Informatics and The Interdependent Energy Infrastructure Simulation System (IEISS). Pattern Informatics, a technique to detect seismic activities and make earthquake predictions, was developed at University of Southern California at Davis. The Pattern Informatics Geographical Information System Grid [1] integrates the Pattern Informatics code with publicly-available, Open Geographical Information System Consortium (OGC)-compatible, geo-spatial data and visualization services. The Interdependent Energy Infrastructure Simulation System is a suite of analysis software tools developed by Los Alamos National Laboratory (LANL). IEISS provides assessment of the technical, economic and security implications of the energy interdependencies [9]. IEISS Geographical Information System Grid, a workflow-style Geographical Information System Grid application developed at LANL, supports IEISS analysis tools by integrating them with openly available geo-spatial data sources and visualization services. Both Pattern Informatics and IEISS systems are in need for an Information Service, which can be utilized as the workflow session metadata manager.

1.3.2.2 A metadata catalog service component

Description: A Metadata Catalog Service is responsible for providing an access/store interface to both prescriptive and descriptive metadata about services.

Requirements: Geographical Information Systems based Grid applications are comprised of various archival services, data sources, and visualization services. Services such as the Web Map [10] and Web Feature [11] service, because they are generic, must provide additional, descriptive metadata in order to be useful. The problem is simple: a client may interact with two different Web Feature Services in exactly the same way (the WSDL is the same), but the two Web Feature Services may hold different data. One, for example, may contain GPS data for the Western United States while the other has GPS data for Northern Japan. Clients must be able to query information services that encode (in standard formats) all the necessary information, or metadata, that enables the client to connect to the desired service. Thus, we see the need for a metadata catalog service, which would manage metadata associated to all these Grid/Web Services, and make them discoverable. A client should be able to get "capabilities" metadata file either from the service itself or from the metadata catalog. Thus, these metadata catalog services are also expected to have a dynamic metadata retrieval capability, which enables the system to dynamically retrieve the capability metadata file from the service under consideration.

Usage Scenario: The two aforementioned application use domains: Pattern Informatics and IEISS Geographical Information System application are comprised of various data and map generating Grid/Web services. Thus, both of these application domains are in need of a metadata catalog service, which would provide a unified and systematic way to find a service through a registry of services.

1.3.2.3 A metadata caching component

Description: This component is responsible for preserving various dynamic metadata generated during a session.

Requirements: In computational grid portals, we see a need for persistent preservation of dynamic metadata. For an example, when users upload their input data to execute a scientific application, the input data is usually given through input form pages, which are tedious to fill out. In most cases, the users will have minor changes on the input parameters to a particular job and resubmit it later. So, the input data must be preserved as metadata in a persistent third-party data store to be reused later in user-system interaction. For another example, some job runs may take hours or days to execute. In order to keep track of dynamically generated session information regarding a running job, some level of persistence is required. Thus, we see the need for a metadata service to provide a persistence storage capability. Here, session metadata can be stored in parent-child relationships. One should be able to create a hierarchical session tree where each branch can be used as an information holder for dynamic metadata with similar characteristics. This would enable the system to be queried for metadata associated to a session under consideration.

Usage Scenario: The Virtual Laboratory for Earth and Planetary Materials (VLab) [12] is a National Science Foundation funded interdisciplinary project which is a Grid/Web Service based system for enabling distributed computational chemistry and material science application for the study of planetary materials. One of the issues that the VLab project is addressing is the preservation of user input data, information about job status and so forth. In order to keep track of such information, a session bean (i.e. a Java Bean Object) is used. As the session is susceptible to system crashes or web-server restart, the serialized form of session beans must be stored in a persistent metadata store. To this end, the VLab project needs a WS-Context Service, a lightweight, Web Services based metadata system to provide persistent storage for the dynamic metadata generated during a session.

1.3.2.4 Context-store for high performance SOAP

Description: A Context-store component is a metadata service responsible for storing redundant/unchanging parts of SOAP messages exchanged in service communication.

Requirements: The SOAP message enables applications on heterogeneous platforms to interoperate with each other by defining text-based remote procedure call (RPC) mechanism. However, the verbose nature of a SOAP message holds potential overheads. For example, when data is converted to and from a SOAP message, both size and processing time of the message is increased substantially. This creates performance inefficiencies in some application domains, such as mobile computing. The mobile computing environment, which holds many physical constraints like limitations in processing power, battery life, and wireless connections, needs an efficient solution to the problem of expensive processing cost of SOAP messages. The redundant/unchanging parts of a SOAP message are XML elements which are encoded in every SOAP message exchanged between two services. These XML elements can be considered as dynamic metadata associated to a conversation. Such metadata can be considered as rarely changing and has a lifetime bounded with duration of the session. In order to achieve optimized Web Service communication, which is most needed in mobile environments due to high communication latency, there is a need for a metadata service to store the redundant XML elements of messages.

Usage Scenario: The Handheld Flexible Representation (HHFR) is an application designed to provide efficient and optimized message exchange paradigm in mobile Web Service environment [13, 14]. The HHFR architecture provides layers, which optimize and stream messages to achieve high performance mobile Web Service communication. The HHFR system needs a third-party repository, (i.e. Context-store) to store the redundant/unchanging parts of the messages exchanged between services. This way the size of the exchanged messages can be reduced to achieve optimized Web Service communication.

1.3.2.5 Managing Real-Time Session Metadata

Description: A Session Metadata Manager component is responsible for managing dynamic metadata generated during audiovisual sessions.

Requirements: Collaborative audio/video sessions may have varying types of metadata describing the group of participants, clients as well as the associated media services. Such metadata can be investigated as static and dynamic. For example, the number of available sessions and their associated detailed information is static in nature, while, participant entities, streams, services or filters involved in a session is dynamic. For real-time audio/video conferencing applications, dynamically changing information should be managed by a third-party metadata repository. This way, the system can keep track of audio/video streams.

Usage Scenario: The Global Multimedia Collaboration System (GlobalMMCS) project [3] is a service-oriented multimedia collaboration system that mainly process varying multimedia streams such as audio, video and so forth. GlobalMMCS multimedia sessions generate real-time metadata describing various entities of a session such as streams. The GlobalMMCS project is in need for a Session Metadata Manager component to provide access/store/search interface to dynamic metadata generated in real-time conferencing applications.

1.4 Contribution

The main contribution of this thesis is to propose an architecture for an Hybrid Grid Information Service supporting both distributed and centralized paradigms and managing both dynamic and slowly varying quasi-static metadata.

The implications of this thesis are seven-fold.

- Identifying the information management requirements of dynamic Grid/Web Service collections (a modest number of dynamic collections of actively interacting Grid/Web Services that are put together for particular functionality) [15-18].

- Proposing an extended version of the existing UDDI Specification to provide a domain-independent and metadata-oriented management of service metadata [19-21]. An example implementation [22, 23] of the proposed specification is presented particularly to meet with the information requirements of the Geographical Information Systems. This implementation presents an approach on how to aggregate and search geospatial services using UDDI and has been used and tested in Geographical Information System application use domain discussed in Section 1.3.2.2.
- Proposing a data model and communication protocol for the Context Manager component of the WS-Context Specifications to provide dynamic, session-related metadata management. This work introduces an efficient mediator Information Service to achieve service communication among interacting Grid/Web Services [19, 21, 24].
- Proposing a novel architecture for fault tolerant and high performance Grid Information Services linking publish-subscribe based messaging schemes with associative shared memory platforms for metadata management [17, 25]. As an example of the proposed architecture, a prototype implementation is presented and evaluated. This implementation utilizes publish-subscribe based messaging infrastructure to implement replication, request distribution and consistency enforcement aspects of a distributed system. Section 1.3.2 discusses various motivating application use domains where the prototype implementation has been used and tested [1, 12, 22, 26-30].

- Proposing a novel architecture for unification of different Grid Information Services under one Hybrid System. This approach introduces a Hybrid Grid Information Service that works as an add-on system above the existing Grid Information Services. It introduces abstraction layers, which enable the system to support one to many information services and their communication protocols.
- Proposing a novel architecture for federation of Grid Information Services in metadata instances. This thesis introduces a common data model and communication language to provide a common platform where customized implementations of Grid Information Services can interoperate and share information. With this approach, we aim to enable different Grid applications to communicate with each other and utilize each other's services.
- Identifying and analyzing the key factors that affect the performance of the information systems with peer-to-peer strategies as well as systems adopting in-memory storage solutions [25].

1.5 Organization of the thesis

This chapter 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, the statement of the research problems is given. In order to present the scope of the research, the requirements expected from this research are outlined. Next, a number of application use domains and their metadata requirements are discussed to emphasize the research problems are worthwhile to answer. Finally, a discussion on the contributions of the thesis is presented.

The organization of the rest of the thesis is as follows. Chapter 2 reviews the major solutions in state of art of the studies covered in this thesis. It analyzes the service metadata under two types: interaction-dependent and interaction-independent. Having identified the two-metadata types, it gives an extensive survey on the previous metadata management solutions under two categories: managing interaction-independent, static metadata and managing interaction-dependent, session-related metadata. Here, previous solutions are analyzed followed by discussions on the reasons why the previous solutions do not answer the research problem at hand. Chapter 2 also discusses various concepts and paradigms that are taken into account in designing a solution addressing the research problem. Chapter 3-5 presents the Hybrid Grid Information Service. Chapter 3 discusses the architectural design details of the system. Chapter 4 gives an overview of the semantics of the system. In this chapter, the two base elements of the semantics of the proposed solution are identified: data model semantics and semantics for XML API. With this identification made, the proposed approach and experiences in designing "semantics" for the Hybrid Service are discussed. Chapter 5 presents the prototype implementation of the system. Chapter 6 analyzes the performance evolution of the Hybrid Service prototype. It presents benchmarking on performance, scalability, distribution, fault-tolerance and consistency enforcement aspects of the system. Chapter 7 contains the thesis summary, answers to research questions and the future research directions.

Chapter 2

Review of State of Art

A Grid/Web Service is a software component that has public programming interface described by XML and is capable of being accessed by using XML based messages passed on by internet protocols [31]. The Computational Grid introduces large amounts of services managed by different organizations or individuals to let users utilize distributed computing resources, applications and data. Peer to Peer computing also provides services where researchers package their own resources as services to offer others in their community. For an example, Geographical Information Systems provide very useful problems in supporting "virtual organizations" and their associated information systems. These systems are comprised of various archival data services (Web Feature Services), data sources (Web-enabled sensors), and map generating services. Organizations like the Open Geospatial Consortium (OGC) [32] define the metadata standards. All of these services are metadata-rich, as each of them must describe their capabilities (What sorts of features do they provide? What geographic bounding boxes do they support?) Furthermore, these services must typically be assembled into short-term service collections that, together with code execution services, are combined into a meta-application (i.e. a workflow). As the services interact (collaborate) with each other in a workflow, they generate metadata, which is the distributed state information. Therefore, we have both stateless and stateful (transient) metadata. This is an example of the very general problem of managing information about Web Services. Thus, we see an emerging need for Information Services managing all kinds of metadata associated to Web Services.

In this chapter, we survey the state of art in this area of investigation. We also overview background knowledge on relevant concepts covered in this thesis such as tuplespaces, publish-subscribe paradigms, replication and consistency issues.

2.1 Managing service metadata as Context

Web Services may have complex characteristics and interact with one or a set of services. Service descriptions expressing these characteristics must be capable of accurately representing these services.

We use the term “context” to define all available information associated with a Web Service. For the purposes of our research, context is a piece of information (metadata) describing behavior, environment and characteristics of a service. Context encapsulates not only activities that service is involved in but also the service itself as an entity. From this point forward, we will be using context and service metadata interchangeably, as they both refer to the information associated with a service.

We broadly classify context into two categories: interaction-dependent and interaction-independent. Interaction-dependent context is the session metadata generated by one or more services as a result of their interactions.[1] Interaction-independent context is rarely changing information describing the characteristics of services.[2]

Another way of classifying context could be based on its characteristics such as prescriptive (functional) and descriptive (non-functional). The prescriptive characteristics are directly related with functionality of the service. For instance, the Open Geographical Information Systems Consortium defines standards for prescriptive characteristics of services as an auxiliary capability file defining the data coverage of geospatial services. The descriptive characteristics are the non-functional properties associated with services. The non-functional properties of services may include availability (such as temporal, spatial availability), service quality (such as throughput, number of max supported clients), security and so forth.

Locating resources of interest is a fundamental problem in resource intensive environments. An effective methodology to facilitate resource discovery is to provide and manage information about resources. Here, a resource corresponds to a service and information associated to it refers to metadata of a service. Thus, we see a greater need for metadata management solutions to make such metadata available in peer-to-peer/grid environments. Having identified the two-metadata types, that is interaction-independent and interaction-dependent; we survey previous solutions for metadata management under two categories: a) managing interaction-independent service metadata and b) managing interaction-dependent service metadata.

[1] An example XML document representing an interaction-dependent metadata is given in Appendix B.1.
[2] An example XML document representing an interaction-independent metadata is given in Appendix B.2.

2.1.1 Managing interaction-independent service metadata

Previous solutions addressing the interaction-independent metadata discovery and storage problem has mainly focused on four different areas. First area covers problems in the **matchmaking process**. The matchmaking process compares the service metadata (i.e. information associated to a service) with an access request (i.e. inquiry constructed by the requesters), tries to match them and produces results. Second area focuses on **centralized and decentralized storage architectures**. The third area focuses on the ways of handling metadata **request distribution** based on underlying networks. Finally, the fourth area focuses on **standardizations** defining interaction-independent service metadata.

2.1.1.1 Analysis of service metadata management research from matchmaking processing point of view

The service matchmaking process is a retrieval process that finds results by matching a service request (inquiry criteria) with service descriptions (metadata). We broadly classify existing research under two major trends: syntactic-level matching and concept-based matching.

- **Syntactic Level Matching:** In this research trend, retrieval is founded on keyword-based (UDDI [8], Corba Naming [33]), unique identifier based (Blootooth [34]), interface-based (JINI [35]), or attribute based matching (Salutation [36], OGSA [37]). These methodologies suggest a syntactic level matching between the access request and service metadata and have their own merits in simplicity of implementation. **Limitations:** Different keywords/attributes might have the same meaning. Likewise, same keywords/attributes might have different meanings. Therefore, these methodologies suffer from syntactical mismatches, which in turn cause poor search results.

- **Concept-based Matching:** Concept-based retrieval [38-40] provides a common data format for both service providers and service requestors. It defines ontologies and provides service matchmaking on concepts as opposed to keywords. This way the limitations of syntactical matchmaking are avoided and better precision and recall can be achieved in the results. An example concept-based retrieval mechanism, ServoGrid Metadata Discovery system [41] utilizes ontologies of an earthquake simulation grid for data representation and provides a retrieval tool for earth scientists to locate resources (codes, data) of interests. Here metadata can be represented using varying metadata models such as Semantic Web languages RDF [42] and OWL [43, 44]. **Limitations:** Defining and creating ontology of a given metadata domain may not be trivial, as it is difficult to bring together scientists agree on a consistent ontology capturing all the concepts of the domain. Say, there are more than one ontology for a given metadata domain. In that case, it may be difficult to combine these ontologies, if they have contradicting concepts describing the same thing.

- **Discussion:** These methodologies mainly focus on improving recall/precision in order to improve the quality of search results. This research has been investigated in [40, 45, 46] and so not covered in this thesis. We view architectural design issues and distributed system aspects of managing metadata as higher priority.

2.1.1.2 Analysis of service metadata management research based on architectural design issues

Existing service metadata discovery architectures can be broadly categorized as centralized and decentralized by the way they handle with service information storage.

- **Centralized Registries:** In centralized approach, there is a central look-up mechanism where all services are dependent on one node. Mainstream service discovery architectures like JINI [35], Salutation [36], and Service Location Protocol [47] have been developed to provide discovery of remote services residing in distributed nodes in a wired network. Their architectures are based on a central registry for service registration and discovery. **Limitations:** The centralized registry approach presents a single point of failure and is limited to a certain storage capability. It does not scale up to high number of services that in turn creates a performance and scalability bottleneck for the system.
- **Decentralized Registries:** In decentralized approach, there is no central database. This research trend mainly focuses on decentralized search where all the peers of the system actively participate the discovery process. Peer-to-Peer systems may broadly be categorized as pure and hybrid [48, 49]. On one hand, pure systems endeavor for total decentralization and self-organization, on the other hand hybrid systems have some form of centralized control. Pure peer-to-peer networks may further be categorized as a) structured and b) unstructured. In structured peer-to-peer architectures, system resource placement at peers is enforced with strict constraints. For an example, Globus Monitoring and Discovery System (MDS4) [50] has a structured architecture where there is a single top-level information service that presents a uniform interface to clients to access data, while the data is collected by lower-level information providers. Relational Grid

Monitoring Architecture (R-GMA) [51] presents a relational model where users query/store/access metadata centrally and if information is found, directly connect to information providers to retrieve the data without intermediary nodes. Another example of the structured peer-to-peer architectures, is the systems where the nodes are equally enabled and controlled and service information is disseminated to all nodes (CAN [52] and Chord [53]). In unstructured peer-to-peer architectures, there is complete lack of constraints on the placement of resources and the capabilities of the system nodes [49]. Each node forwards the incoming query to a neighbor based on a routing strategy. An extensive survey on Grid Information Services can be found at [6, 7].

Architectures with pure decentralized storage models have focused on the concept of distributed hash tables (DHT) [52, 53]. The DHT approach assumes possession of an identifier such as hash table that identifies the service that need to be discovered. Each node forwards the incoming query to a neighbor based on the calculations made on DHT. For instance, a DHT specifies a relation between a resource and a position in a distributed network. A good example of DHT is the Chord [53] project. Each entity of the network is hashed; therefore, the position of the entity in the network is determined through DHT. A message is routed to the closest entity to the final destination. Inspired from peer-to-peer discovery model, there has been work conducted on to develop peer-to-peer architectures [54, 55] for distributed information management.

Another decentralized approach, Bittorent is a peer-to-peer file distribution protocol which is designed to distribute large amounts of widely distributed data. A Bittorent network consists of three entites: a tracker, a torent file and peers. A tracker is a server that keeps track of which peers (seeds, downloaders) are in the network. A torent is a metadata file that contains information about all the downloadable pieces of a data. A peer is software, which implements the Bittorent protocol. Each peer is capable of requesting, and transferring data across network. Peers are classified into two categories: seeds and downloaders. The former has the complete copy of the file and offers it for download. The latter has the parts of the file and downloads the file from other seeds or downloaders. An example peer-to-peer storage service, Amazon Simple Storage Service (Amazon S3) is a web-scale storage which supports use of the Bittorent protocol. It provides a simple web service interface used to provide storage and retrieval of any data across widely distributed area.

The data management in decentralized systems is mainly studied by distributed database systems research [56]. This research area enables applications to share data at a higher conceptual level, while ignoring the implementation details of the local data systems. In turn, this enables transparent access to multiple, logically interrelated distributed databases. To achieve this, a distributed database system, which allows management of database systems with different schemas, is defined [57]. 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. The distributed database systems achieve this transparency by providing a schema management. The schema management can be achieved with either a centralized approach or a decentralized schema mapping approach. The former approach defines a global schema over the existing data sources and mappings between global schema and local database schemas. The latter approach maps a query on a given data system schema to another query of another data system's schema.

Limitations: As the resource placement at nodes is strictly enforced in structured peer-to-peer networks, these systems suffer from a heavy overhead on the bootstrap of the network. Pure decentralized storage models have mainly focused on DHT approach. The DHT approach provides good performance on routing messages to corresponding nodes. However, it is limited to primitive query capabilities on the database operations [48]. Furthermore, the DHT approach does not take into account changes in the client demands and load balancing. The Bittorent approach and Amazon S3 storage service that utilizes the Bittorent protocol have the following limitations. First, the overhead involved in transferring small size data (e.g. in the order of kilobytes) is big. For example, the total required bandwidth for necessary protocol messages for downloading a small size data is high. Second, the tracker is a performance bottleneck and a single point of failure in the network. Thus, the performance of a Bittorent network depends on the capacity of the tracker. In addition, if the tracker fails, it is not possible for peers to locate each other. To achieve data integration, centralized or decentralized schema mapping approaches can be utilized. The global schema approach captures expressiveness capabilities of customized local schemas. However, this approach cannot scale up to high number of data sources. The decentralized schema mapping approach is able to express high-level queries over customized data-system schemas without relying on a global schema; however, this approach limits the query expressiveness.

- **Discussion:** The centralized storage scales better in performance for limited storage capability compared to decentralized approach, whereas a decentralized approach can scale up to high amount of metadata where centralized approach fails. Pure decentralized storage models such as peer-to-peer service discovery architectures have focused on the concept of distributed hash tables (DHT). This method may provide better performance as the database operation messages are routed fast, however, it still does not provide the same performance to handle dynamic metadata as centralized database does. The research ideas in distributed database systems can be revisited to achieve information integration in Grid Information Services. The distributed database systems enable information integration through query processing. In other words, it 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. To achieve high performance, there is a need for a higher-level add-on architecture that can assemble the information coming from different data sources and carries out queries on the heterogeneous information space. We think that once we achieve such higher-level architecture, the global schema approach can be used for integrating a limited number of widely used information service schemas, as it encapsulates the expressiveness power of the customized schemas that are being integrated.

In this thesis, we will take as a design requirement that the proposed system should link peer-to-peer and centralized metadata storage strategies. The proposed system should be designed to provide a) management for small-size metadata, b) high performance by utilizing in-memory storage solutions, c) fault-tolerance by increasing the availability of metadata, and d) peer-to-peer message distribution strategy by utilizing a classic middleware approach; publish-subscribe based messaging system. The proposed system should be designed as an add-on architecture above existing Grid Information Services. It should also be designed to provide unification and federation of information coming from different sources under one hybrid system. To achieve this, global schema approach can be revisited to achieve a unified schema integrating different Grid Information Service Schemas.

2.1.1.3 Analysis of service metadata management research based on formation of underlying networks

Another way of classifying service discovery architectures could be based on the formation of the network and the way of handling with discovery request distribution.

- **In traditional wired networks**, network formation is systematic since each node joining the system is assigned an identity by another device in the system [58, 59]. Example wired network discovery architectures such as JINI [35] and Service Location Protocol [47] focus on discovering local area network services provided by devices like printer.
- **In ad-hoc networks (unstructured peer-to-peer systems)**, there is no controlling entity and there is no constraint on the resource dissemination in the network. Existing solutions [58, 60] for service discovery for ad-hoc networks (e.g. pervasive computing environments) can be broadly categorized as broadcast-driven and advertisement-driven approaches [61]. In broadcast-driven approach, a service discovery request is broadcasted throughout the discovery network. In this approach, if a node contains the service, it unicast with a response message. In advertisement-driven approach, services advertise themselves to all available nodes. In this case, each node interested discovering a service caches the advertisement of the service. The WS-Discovery Specification [62] supports both broadcast-driven and advertisement-based approaches. To minimize the consumption of network bandwidth, this specification supports the existence of registries and defines a multicast suppression behavior if a registry is available on the network.

Limitations: The traditional wired-network based architectures are limited, as they depend on a controlling entity, which assigns identifiers to participating entities. If the size of the network is too big, the broadcast-driven approach has a disadvantage, since it utilizes significant network bandwidth, which in turn creates a large load on the network. The advertisement-driven approach does not scale, as the network nodes may have limited storage and memory capability. The WS-Discovery approach is promising to handle metadata in peer-to-peer computing environment; however, it has the disadvantage of being dependent on hardware multicast for message dissemination.

- **Discussion:** Metadata discovery solutions designed for ad-hoc networks are appropriate for Grid and peer-to-peer computing environments, as these solutions do not have any constraints on resource dissemination in the network. Among these solutions, the WS-Discovery approach is promising as it employs a pure peer-to-peer approach where the messages (advertisement/discovery) are broadcasted in the system.

Inspired by WS-Discovery approach, we will take as a requirement that the proposed system should employ a broadcast-based metadata discovery approach. Each message should include a unique identifier distinguishing the peer, which initiated the request. On receipt of a message, only the nodes that have the requested information should reply with a response message. Moreover, we will also take as a requirement that the proposed system should employ an advertisement-driven approach for advertising the existence of network nodes. Apart from the WS-Discovery approach, the proposed system should use a software multicast based message dissemination for request distribution, metadata and network node advertisements.

2.1.1.4 Specifications defining interaction-independent service metadata

As the Service Oriented Architecture (SOA) principles [63] have gained importance, an emerging need has appeared for standardization of XML metadata services that provide programming interface to access and manipulate service metadata. The previous sections introduced the concept of "context" as the service metadata and surveyed previous solutions that provide management/discovery of rarely changing, interaction-independent metadata. This section investigates the existing specifications/ standardizations defining the service metadata. In our investigation, we mainly focus on metadata requirements of Geographical Information Systems, as they provide very useful problems in supporting "virtual organizations" and their associated information systems.

• **Web Registry Services:** The Web Registry Service [64], introduced by the Open Geographical Information Systems Consortium (OGC) [32] is an approach to standardize the metadata management problem particularly for Geographical Information Systems domain. The OGC is an international organization providing specifications to integrate geospatial data and geo-processing resources into mainstream computing. It leads efforts to provide a) standardized protocols for accessing geospatial information and services and b) standardized service metadata such as "capabilities.xml" documents. The OGC introduced a) the Catalog Specification [65] and b) the Web Registry Service (WRS) Specification [64]. The OGC Catalog Specification is an abstract specification, which was introduced to create a conceptual model to allow the creation of implementation specifications for discovery and retrieval of metadata that describes geospatial data and geo-processing services. The Web Registry Service (WRS) Specification is an implementation specification of the OGC Catalog Specification, which was introduced to define a standard way to discover/publish service information of geospatial services and presents a domain-specific registry capability for geospatial information. The WRS Specification adopts the OGC Registry Information Model, which is based on the ebXML registry information model (ebRIM) [66, 67]. The WRS Specification uses ebRIM to support/integrate service entries with metadata and provide metadata management for geospatial domain. An example prototype [68] of the WRS Specification is implemented by LAITS group in George Mason University. This prototype is primarily based on Metadata Catalog Service (MCS) [69], a stand-alone metadata catalog service with an Open Grid Service Architecture (OGSA) [37] service interface. The prototype implementation provides a mapping between the OGC Registry Information Model and the MCS data model. **Limitations:** The WRS approach is limited to the Geographical Information Systems domain. As it was designed as a Geographical Information Systems domain-specific solution, it supports neither the information model nor the programming interface that could facilitate a generic metadata management.

- **UDDI:** The Universal Description, Discovery, and Integration (UDDI) Specification is the most prominent and widely used standard that is based on a XML-based protocol that provides a directory and enables services advertise themselves and discover other services. UDDI is 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. As it is WS-Interoperability (WS-I) [70] compatible, UDDI has the advantage being interoperable with most existing Grid/Web Service standards. **Limitations:** We observe that the adoption of UDDI Specification in various domains such as Geographical Information Systems is slow, since the existing UDDI specification has following limitations. First, UDDI introduces keyword-based retrieval mechanism. It does not allow advanced metadata-oriented query capabilities on the registry. Second, UDDI does not take into account the volatile behavior of services. Since Web Services may come and go and information associated with services might be dynamically changing, there may be stale data in registry entries [71]. Third, since UDDI is domain-independent, it does not provide domain-specific query capabilities such as geospatial queries. Thus, UDDI should be extended to overcome these limitations.

- **OGC use of UDDI Registries:** In order to remedy some of these limitations, various solutions have been introduced. For an example, OGC has proposed a set of design principles, requirements and spatial discovery methodologies for discovery of OGC services through UDDI interface [72]. The proposed methodologies have been implemented by various organizations such as Sycline [73] and Galdos [74]. 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 [72]. Briefly, these methodologies showed that it is possible to do spatial discovery and content discovery through UDDI Specification. **Limitations:** Existing UDDI approaches by OGC community are designed for and limited to geospatial specific usage. Services such as the Web Map and Web Feature service, because they are generic, must provide additional, descriptive metadata, such as Quality of Service attributes, in order to be useful. OGC approach does not define a data model rich enough to capture descriptive metadata that might be associated with service entries. That is, their approach does not put the descriptive metadata in the UDDI Registry. Therefore, it is still an open problem how to make these geospatial services distinguishable from others based on their qualities. Thus, we see the need for extensive metadata-oriented query capabilities in addition to geospatial query capabilities. We also note that the discovery methodologies (introduced by OGC community) extend the UDDI interface; however, they do not introduce an extension to existing UDDI information model.

- **UDDI-Extensions:** The UDDI-M [75] and UDDIe [71] projects introduce the idea of associating metadata and lifetime with UDDI Registry service descriptions where retrieval relies on the matches of attribute name-value pairs between service description and service requests. UDDI-M^T [45, 76] improves the metadata representation from attribute name-value pairs into RDF triples. A similar approach to leverage UDDI Specification was introduced by METEOR-S [77] project which identifies different semantics when describing a service, such as data, functional, quality of service and executions. Another approach, Grimories [78] is also an implementation of UDDI Specification. The Grimories Registry extends the functionalities of UDDI to provide a semantic enabled registry designed and developed for the MyGrid project [79]. It supports third-party attachment of metadata about services. The Grimories represents all published metadata in the form of RDF triples allows the published metadata reside either in a database, or in a file, or in a memory. **Limitations:** 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. (How can registries facilitate geo-spatial queries on a metadata catalog for Geographical Information Systems domain?) We note that, the Grimories approach utilizes a caching mechanism for the RDF triple store. Although, this is a promising approach, we find the following limitations. Firstly, the performance of the system is bounded by the performance of the triple store (The Grimories uses the Jena software toolkit to operate on the RDF triple store. Thus, the limitations of Jena implementation may cause a performance bottleneck). Secondly, if the memory was chosen as the primary storage, the Grimories registry would sacrifice persistency as the snapshots of memory are not backed-up by the system. If the database was chosen as the primary storage, the system would sacrifice performance, as the system has to make disk access to publish a metadata. Thirdly, the Grimories's memory built-in storage does not provide mutual exclusive access to the shared data.

- **Discussion:** This thesis investigates methodologies, compatible with widely used standards, for discovering services based on both general and domain-specific search criteria. An example for domain-specific query capability is Xpath queries on the auxiliary and domain-specific metadata files stored in the UDDI Registry. Another distinguishing aspect of our investigation is the support for session metadata.

We will take as a requirement that our system should support not only quasi-static, stateless metadata, but also more extensive metadata requirements of interacting systems. Similar to existing solutions (UDDI-M and UDDIe), the proposed design should use name-value pairs to describe characteristics of services and extend UDDI's Information Model to associate metadata with service descriptions. This approach has its own merits in the simplicity of design and implementation. The proposed system should also explore an in-memory storage mechanism that would provide persistency, performance and data sharing capabilities all together. UDDI-M^T and METEOR-S are example projects that utilize semantic web languages to provide better service matchmaking in retrieval process. This research has been investigated [45, 76, 77] and so not covered in our investigation. We view dynamic and domain-specific metadata requirements of sensor/ Geographical Information System and collaboration Grids as higher priority.

2.1.2 Managing interaction-dependent service metadata

Often Web Services are assembled into short-term service collections that are gathered together into a meta-application (such as a workflow) and collaborate with each other to perform a particular task. For example, an airline reservation system could consist of several Web Services, which are combined together to process reservation requests, update customer records, and send confirmations to clients. As these services interact with each other, they generate session state, which is simply a data value that evolves as result of Web Service interactions and persists across the interactions. As the applications, employing Web Service oriented architectures, need to discover, inspect and manipulate state information in order to correlate the activities of participating services, an emerging need appeared for the technologies and specifications that would standardize managing distributed session state information. We can broadly classify existing solutions that define the stateful interactions of Web Services under two categories: a) point-to-point and b) third-party.

- **Point-to-Point methodologies to enable service communication:** Point-to-point methodologies provide service conversation with metadata from the two services that exchange information. There are varying specifications focusing on point-to-point service communication, such as Web Service Resource Framework (WSRF) [80] and WS-Metadata Exchange (WS-ME) [81]. WSRF specification, which is proposed by Globus alliance, IBM and HP, defines conventions for managing state, so that collaborating applications can discover, inspect, and interact with stateful resources in standard and interoperable ways. The WS-ME provides a mechanism a) to share information about the capabilities of participating Web Services and b) to allow querying a WS Endpoint to retrieve metadata about what to know to interact with them. **Limitations:** Point-to-point methodologies provide service conversation with metadata only from the two services that exchange information.
- **Third-party methodologies to enable service communication:** Communication among services can be achieved with a third-party based metadata management strategy. The Web Services Context Specification (WS-Context) [82] is a promising example of this trend. It was introduced as a part of the Web Services Composite Application Framework (WS-CAF) [83] which is a suite of three specifications, WS-Context, WS-Coordination Framework (WS-CF) [84], and WS-Transaction Management (WS-TXM) [85]. The WS-Context defines a simple mechanism to share and keep track of common information shared between multiple participants in Web Service interactions. WS-CF defines a coordinator to which Web Services are registered to ensure messages and results are communicated correctly. The coordinator provides the notification of outcome messages to Web Services participating in an activity. WS-TXM defines three distinct

transaction protocols: two phase commit, long running actions, and business process flows. These are used in the coordination framework to make existing transaction managers interoperable. The three specifications comprise a stack of functionality [83]. WS-Context is at the bottom and adding WS-CF and then WS-TXM.

The WS-Context 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. For an example, a participating application can discover results of other participants' execution, which is stored as context. 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.

The WS-Context and UDDI introduce two different ways of managing service metadata. The WS-Context defines a standard way of maintaining distributed session state information associated to participating services. The UDDI is a standard way of publishing/discovering generic information associated to Web Services. Therefore, the two-metadata management solutions – UDDI and WS-Context – are comparable, as they, both deal with service metadata. Firstly, the UDDI is concerned with the interaction-independent metadata space. The interaction-independent metadata is rarely changing information describing functional or non-functional properties of Web Services. On the other hand, the WS-Context is concerned with the interaction-dependent metadata space. The interaction-dependent metadata is highly updated and dynamic information describing information associated to Web Service activities. Thus, the two-metadata services define different functionalities to meet the requirements of the two different metadata domains. Secondly, in the WS-Context approach, the members of an activity should be notified of the distributed state information such as when it is created or deleted. This way, the dynamism in the metadata is captured by the participating services of the activity. However, in the UDDI approach, the interaction-independent metadata is rarely changing and may not necessarily require a notification mechanism. Thirdly, the WS-Context is intended for activities that are comprised of modest number interacting Web Services. However, the UDDI is intended for the whole Grid. Thus, the UDDI requires a degree of complexity in inquiry operations to improve the selectivity and increase the recall and precision in the search results. Fourthly, the WS-Context is intended to correlate activities of Web Services that participate to an activity. Thus, it supports loose coupling of services by employing synchronous callback facilities. However, the UDDI is a synchronous Web Service and provides an immediate response to a query. Fifthly, the WS-Context approach should be lightweight for allowing multiple Web Services to share a common context. Thus, it requires high performance and scalability in numbers for concurrent accesses. However, in the UDDI approach, Web Service metadata entry can only be updated by its publisher and is not shared, thus concurrency is not a high priority. **Limitations:** We find various limitations in WS-Context Specification in supporting stateful interactions of Web Services. First, 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. Second, 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:** Among the existing specifications, which standardize service communications, we believe that the WS-Context Specification is the most promising to tackle the problem of managing distributed session state. Unlike the other service communication specifications, WS-Context models a session metadata repository as an external entity where more than two services can easily access/store highly dynamic, shared metadata.

Thus, we will take as a design requirement that the proposed system should utilize an extended version of WS-Context Specification to manage dynamically generated session metadata. In order to remedy the limitations of WS-Context, the proposed approach should support a fault-tolerant, high-performance Hybrid XML Metadata Service.

2.2 Publish-Subscribe Paradigm

Most distributed systems rely on passing messages between processes. Thus, system entities communicate with each other by exchanging messages, which captures varying information such as search/storage requests, system conditions and so forth. These systems can be categorized based on their messaging infrastructures such as publish-subscribe systems, point-to-point communication systems, queuing systems, and peer-to-peer based systems [86]. Among them, publish-subscribe paradigm principles have gained importance in recent years, as recently released specifications such as Java Message Service [87] and WS-Eventing Specification [88] benefit from publish-subscribe system principles to standardize development of interoperable systems. The publish-subscribe paradigm uses an asynchronous messaging. In a publish-subscribe system, publishers can broadcast each message (e.g. through a topic), rather than addressing it to specific recipients. The messaging system then sends the message to all recipients that subscribed to a topic. **Advantages:** As it is asynchronous, a publish-subscribe system forms a loosely coupled architecture where the publishers do not know who the subscribers are. This messaging scheme is more scalable architecture than point-to-point solutions, since message senders only deal with creating the original message, and can leave the job of message distribution to the messaging infrastructure. **Limitations:** Messages are typically broadcasted over a network. This allows a more dynamic network topology. However, as the volume of messages increase, this may result in overloading of the network without appropriate pruning strategies.

Discussion: We will take as a requirement that our system should support the publish-subscribe paradigm as a communication middleware for message exchanges between system entities.

NaradaBrokering [89-93] is an open-source and distributed messaging infrastructure implementing the publish-subscribe paradigm. It establishes a hierarchy structure at the network, where a peer is part of a cluster that is a part of a super-cluster, which is in turn part of a super-super-cluster and so on. The organization scheme of this scenario forms a communication between peers that increases logarithmically with geometric increase in network size. The NaradaBrokering software is the most appropriate solution for our design decision, since its entities, i.e. brokers, specify constraints on the quality of service related delivery of events. It provides a substrate of Quality of Services (security, reliability, etc.). In turn, this enables various capabilities to the system such as order, duplicate elimination, reliable message delivery, security and so forth. Note that these capabilities are not inherently part of publish-subscribe paradigm.

2.3 TupleSpaces Paradigm

A TupleSpace forms a 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 [94] 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 [95] project by Sun extends and implements Linda. Likewise, IBM has a tuplespaces implementation called TSpaces [96]. 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 [97]) and Semantic Web (Triple Spaces [98], Semantic Web Spaces [99]). **Advantages:** The tuplespaces concept provides the ability for data sharing and coordinating events of processes. It enables processes exchange/share data regardless of whether their lifetime overlaps. It also enables mutual exclusive access on the shared data. This in turn provides coordination of events of processes. **Limitations:** The tuplespaces paradigm is a centralized solution and exposed to limitations of centralized systems such as single-point of failure and performance bottleneck.

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 consider tuplespaces paradigm as an appropriate model to enable communication between Web Services. We will take as a requirement that our design should employ the tuplespaces paradigm as an in-memory storage. The decentralized approaches can scale up to high amount of metadata. Therefore, this thesis should also investigate how to link a centralized in-memory approach with decentralized peer-to-peer systems to provide an approach for a Grid Information Service. A java implementation of the TupleSpaces concept, JavaSpaces [95], was released by Sun MicroSystems [100]. 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 [101], 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 will 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.

2.4 Replication and Consistency Issues

Replication is a well-known and commonly used technique to improve the quality of metadata hosting environments. One approach to replication is to keep a copy of a data at every node of the network (full replication). The other approach is to keep a copy of a data only at a few number of replica servers (partial replication) [102, 103]. Replication can further be categorized as permanent-replication and server-initiated replication [103]. Permanent-replication keeps the copies of a data permanently for fault-tolerance reasons, while the server-initiated replication creates the copies of a data temporarily to improve the responsiveness of the system for a period of time during which the data is in high demand.

Sivasubramanian et al [102] give an extensive survey on designing and developing replica hosting environments, as does Robinovich in [104], paying particular attention to dynamic replication. As the nature of some of the targeted metadata domains of this thesis is highly dynamic, we focus on replica hosting systems that are handling with dynamic data. These systems can be discussed under following design issues: a) distribution of client requests, b) selection of replica servers for replica placement, and c) consistency enforcement.

Distribution of client requests is the problem of redirecting the request to the most appropriate replica server. Some of the existing solutions to this problem rely on the existence of a DNS-Server [104, 105]. These solutions utilize a redirector/proxy server that obtains physical location of a collection of data-systems hosting a replica of the requested data, and choose one to redirect client's request.

Replica placement is another issue that deals with selecting data hosting environments for replica placement and deciding how many replicas to have in the system. Some of the existing solutions that apply dynamic replication, monitor various properties of the system when making replica placement decisions [104, 106]. For instance, Radar [107] replicates/migrates dynamic content based on changing client demands. Spread [106] considers the path between the data-system and the client and makes decisions to replicate dynamic content on that path.

The consistency enforcement issue has to do with ensuring all replicas of the same data to be the same. A consistency enforcement model is a contract between a hosting environment and its clients [102]. Some classification approaches to categorize existing research for consistency enforcement are discussed in [102, 103]. Tanenbaum [103] differentiates consistency under two main classes: data-centric and client-centric. In the data-centric approach, all copies of a data are updated regardless of whether some client is aware of those updates. In the client-centric approach, consistency is ensured from a client's perspective. Client-centric consistency model allows copies of a data to be inconsistent with each other as long as the consistency is ensured from a single client's point of view. The implementations of the consistency models can be categorized as primary-based protocols (primary-copy approach) and replicated-write protocols [103]. In primary-copy approach, updates are carried out on a single server, while in the replicated-write approach; updates can be originated at multi servers. For an example, Radar [104] applies the primary-copy approach, which suggests a copy of a data item to be designated as primary-copy, to ensure consistency enforcement. Updates can be transferred in different ways. One approach, for example, is to transfer the whole content of a replica, while the other is to transfer the difference between the previous copy and the updated copy. Update propagation can be initiated in different ways. For example, data may be pulled from an up-to-date server (pull). Another example, an up-to-date server may keep track of the servers holding copies of a data and push the updates onto those servers (push). Some update propagation schemes combine pull and push methodologies. For instance, the Akamai project [105] introduces versioning where a version number is part of the data identifier, so that the client can only fetch the updated data (with a given identifier) from the corresponding data hosting system.

Discussion: The proposed architecture should differ from previous solutions for web replica hosting systems, as the intended use is not to be a web-scale hosting environment. Table 1 shows a summary of the useful strategies that we take as a design requirement for our implementation design.

As for the request routing mechanism, we think that, broadcasting access requests would be the most appropriate request distribution solution considering our targeted domains. Some of the existing solutions to dynamic replication [104, 105] assume all data-hosting servers to be ready and available for replica placement and ignore "dynamism" both in the network topology and in the data. In reality, data-systems can fail anytime and may present volatile behavior, while the data can be highly updated. Thus, to capture such "dynamism", we take as a requirement that the proposed system should broadcast the requests to the nodes holding the data under question. For message dissemination, the system should employ a pure peer-to-peer approach, which is based on publish-subscribe based messaging schemes to achieve a multi-publisher multicast mechanism.

Design Issue	**The design requirements of the proposed system**
Replica-content placement	copies of a context should be kept permanently for fault tolerant reasons (permanent replication)
	copies of a context should be kept temporarily for a time period during which the context is in demand to improve performance (server-initiated replication)
Request routing	client's request should be broadcasted to those nodes holding the context in question (broadcast-based request dissemination)
Consistency enforcement	updates should be carried out on a single server (primary-copy approach) - every update request should be assigned a synchronized timestamp, which can later be used for ordering among the updates
	copies of a context can be inconsistent with each other; however, they should be consistent from a client's perspective.
	whole content of a context should be broadcasted by the primary-copy to the redundant permanent-copy holders

Table 1 Summary of the replication and consistency enforcement strategies that we take as a requirement for the proposed system implementation.

As for the replica placement methodology, we consider providing an architecture, which would allow both partial and full replication to take place with negligible system processing overheads. We also consider both permanent and server-initiated replication as appropriate strategies for the proposed system. The permanent-replication could provide a minimum required fault tolerance, while the server-initiated replication could improve responsiveness of the system.

To minimize the cost of consistency enforcement, we take as a requirement that the system should employ a client-centric consistency model, which suggests copies of a context can be inconsistent with each other; however, they should be consistent from a client's perspective. The proposed approach should provide consistency models addressing consistency requirements of different application domains.

As for the consistency enforcement protocol, the primary-copy approach is the most appropriate solution for the proposed approach based on the requirements of aforementioned application use domains (See Section 1.3.2). In the primary-copy approach, to perform an update operation just the primary-copy is locked. Since primary copies are distributed at various data-systems, a single site will not be overloaded with locking all its data for update operations. Thus, we take as a requirement that the system should support the primary-copy approach at the implementation stage of consistency enforcement.

As for the way an update is initiated, the push approach could be an appropriate solution. The push approach has a disadvantage since it requires the primary-copy host to store and keep track of the state of each replica server holding a copy of the replica. To overcome this limitation, we take as a requirement that the system should introduce an approach, which utilizes broadcast-based dissemination to send updates only to those nodes holding the redundant copies of a context. Based on this scheme, the primary-copy host could push the updates, when an update occurs. This multicast-based approach does not require the primary-copy host to keep the state of the partial replica set of a context.

2.5 Summary

This chapter discussed the state of art in the research area of service metadata management in Information Services. First, an overview of service metadata and metadata management was presented. Then existing solutions are identified under several mainstream categories based on the ways they tackle with the research issues in sub-processes of metadata management: a) service metadata matchmaking processing b) architectural design for storage handling c) formation of underlying networks c) standardizations on service metadata management. Having identified these categories, previous solutions, their advantages and limitations are investigated followed by discussions. From this, we have identified useful strategies that we will use in our architecture. We also overviewed background knowledge on various concepts such as the publish-subscribe paradigm, the TupleSpace paradigm, replication and consistency issues involved in dynamic and distributed metadata management.

Chapter 3

Architecture

Chapter 2 analyzed the existing solutions and their limitations involved in managing context associated to Grid/Web Services. Based on the analysis, this chapter particularly focuses on the modular architecture of a system by addressing the research problem given in Section 1.2 and the limitation of previous solutions discussed in Section 2.1.

3.1 System overview

We have designed a novel architecture for a Hybrid Grid Information Service addressing the metadata management requirements (see Section 1.3.1) of aforementioned application domains (see Section 1.3.2) to support handling and discovery of not only quasi-static, stateless metadata, but also session related metadata [15-17, 25, 108].

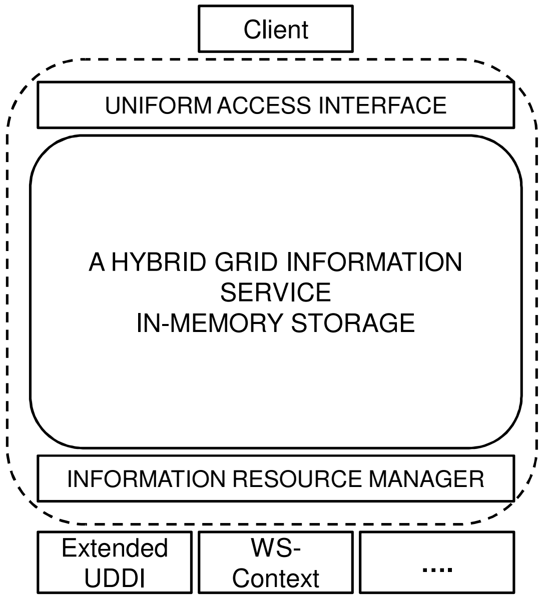

Client
UNIFORM ACCESS INTERFACE
A HYBRID GRID INFORMATION SERVICE IN-MEMORY STORAGE
INFORMATION RESOURCE MANAGER
Extended UDDI
WS-Context
....

Figure 2 This figure illustrates the centralized version of a Hybrid Grid Information Service interacting with a client. The dashed box indicates the Hybrid Service. It is an add-on architecture that runs one layer above information service implementations (such as the extended UDDI XML Metadata Service (our implementation of UDDI Specification) and WS-Context XML Metadata Service (our implementation of Context Manager component of the WS-Context Specification)) to handle metadata associated to services. To facilitate management of service metadata, the prototype integrates both UDDI and WS-Context Specification implementations. It provides a uniform access interface by utilizing UDDI and WS-Context XML API to interact with the clients. It utilizes an Information Resource Manager abstraction layer to interact with lower layer Information Services.

To meet the uniformity requirements, the Hybrid Grid Information Service architecture presents abstraction layers, which enables the system to support one to many information services and their communication protocols. This way, the system unifies different information services under one hybrid system.

To meet the federation requirement, it presents a federation capability where different information services can be federated in metadata instances. To facilitate this capability, we introduce a Unified Schema by integrating different information service schemas. The Unified Schema provides a common platform to enable interaction between customized implementations of Grid Information Services. With this capability, the system allows users to provide their own mapping rules. The Hybrid System provides transformations between metadata instances of the Unified Schema and the customized Grid Information Service Schemas (such as Extended UDDI Schema).

To meet the interoperability requirement, and to be compatible with existing Grid/Web Service standards, we implemented two Information Services (Extended UDDI XML Metadata Service and WS-Context XML Metadata Service) based on WS-I compatible WS-Context [82] and UDDI [8] Specifications. A centralized version of the architecture is depicted in Figure 2. This figure illustrates a client interacting with the Hybrid Grid Information Service, which is running as an add-on component above the Extended UDDI and WS-Context Information Services.

To meet the performance requirement, the Hybrid Grid Information Service utilizes an in-memory storage to minimize average execution time for standard operations. The in-memory storage provides associated shared memory platform and is designed based on the TupleSpaces paradigm (see Section 2.3). This in turn minimizes the access latency.

To provide persistency of information, the Hybrid Information Service backs-up newly-inserted/updated metadata instances into the appropriate information service implementation back-end with certain time intervals. To meet the fault-tolerance requirement (see Section 1.3.1), the system utilizes replication technique to provide high availability of information. Section 2.4 gives a brief overview of existing solutions to replication technique. Replication can be categorized by the manner in which replicas are created and managed. One strategy is permanent replication: replicas are manually created, managed and kept permanently. This strategy is mostly used for fault-tolerance reasons. Another strategy is server-initiated (dynamic) replication: replicas are created, managed and kept based on changing user behavior. This strategy is mostly used to enhance system performance. The Hybrid Grid Information Service utilizes both permanent replication and dynamic replication technique. The permanent replication is used to provide fault-tolerance in terms of availability. The dynamic replication technique is used for performance optimization. Here, the dynamic replication technique enhances performance by replicating data onto servers in the proximity of demanding clients that in turn reduces access latency. Figure 3 illustrates the decentralized version of the architecture.

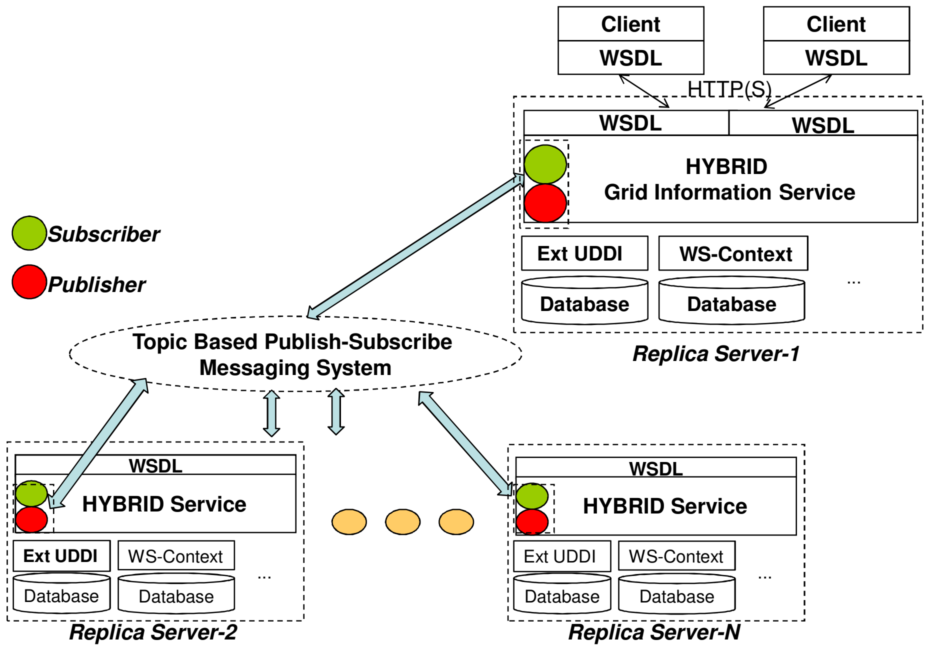


Figure 3 Distributed Hybrid Grid Information Services. This figure illustrates N-node decentralized Hybrid Service from the perspective of a single Hybrid Service (Replica Server-1) interacting with two clients. The Hybrid Grid Information Service uses a topic based publish-subscribe messaging system to enable communication between its instances.

To meet the dynamism requirement (see Section 1.3.1), the proposed system introduces a) efficient metadata access/storage capabilities and b) optimization techniques for self-adaptation to instantaneous client-demand changes. Firstly, to achieve efficiency in metadata access and storage, it utilizes publish-subscribe based messaging schemes. For an example, if a query cannot be granted locally and requires external metadata, the request is broadcasted only to those nodes hosting the requested metadata in the network at least to retrieve one response satisfying the request. This way the system is able to probe the system to look for a running server carrying the right information at the time of the query. Secondly, we observe that in a dynamic Grid/Web Service collection, the nature of data is very dynamic and the replica servers hosting copies of a context may have volatile behavior. Moreover, metadata may have changing user demands. To capture the dynamic behavior, the Hybrid System uses the dynamic replication technique where copies of a data may be created, deleted, or migrated among hosting data-systems based on changing user demands [109].

3.2 Network communication model

An important aspect of the proposed system is that it utilizes a software multicasting capability as a communication medium for sending out access and storage requests to the network nodes. This is a topic-based publish-subscribe software multicasting mechanism, and it is used to provide message-based communication. Any node can publish and subscribe to topics, which in turn create a multi-publisher multicast broker network.

The architectural design of the proposed system is built on top such publish-subscribe based multicast broker network system as depicted in Figure 4. In this illustration, each peer runs a Hybrid Grid Information Service. We use NaradaBrokering [110] publish-subscribe system as a communication middleware for message exchanges between peers. NaradaBrokering establishes a hierarchy structure at the network, where a peer is part of a cluster that is a part of a super-cluster, which is in turn part of a super-super-cluster and so on. It provides efficient and reliable message delivery to the targeted peer en route to intended clients. For example, in Figure 4, we observe various message delivery routes from peer-2 to peer-7. The NaradaBrokering software is able to make decision to choose the most efficient message delivery route, i.e., 2-6-7, as opposed to inefficient delivery routes such as 2-3-5-6-7 or 2-3-4-10-9-8-6-7. Here, every peer, either targeted or en route to one, computes to shortest path to reach target destinations.

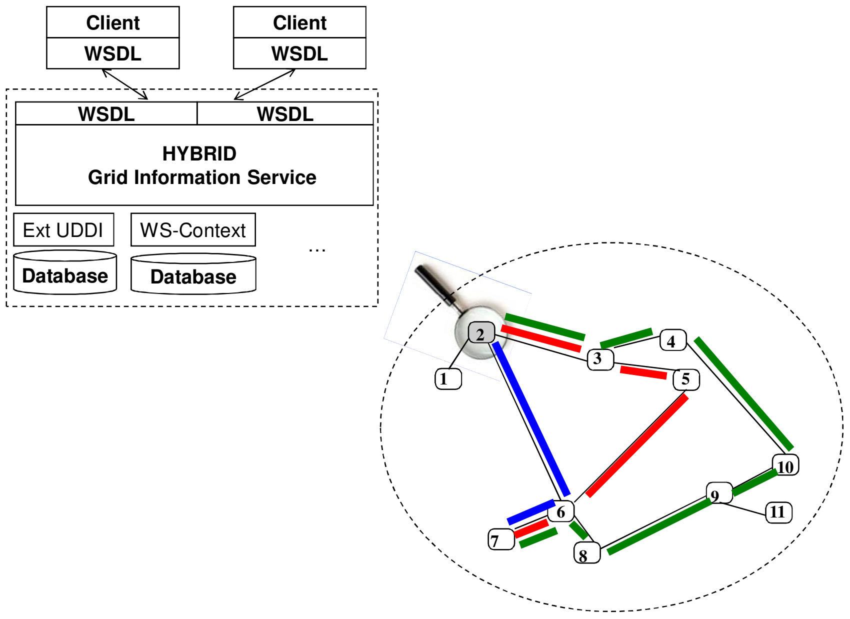


Figure 4 An example eleven-node Hybrid Service metadata hosting environment where each node is connected with publish-subscribe based overlay network. Numbered squares represent nodes running Hybrid Services (see Figure 2 for centralized version of the ser service). The thick lines on the figure are used to show different message delivery routes between peers 2 and 7 that are described in the text.

3.3 Assumptions

Our architectural design relies on following assumptions.

- **Memory Management:** We assume that today's servers are capable of holding both interaction-independent and interaction-dependent metadata (associated to Grid/Web Services) in in-memory storage.
- **Consistency:** A client-centric consistency model (that allows replicas of the same context to be different within the borders of the system, while providing consistency from the perspective of a client) is sufficient for the targeted application use domains described in Section 1.3.2.

3.4 System Components

The architectural design of the proposed system consists of abstraction layers. Figure 5 illustrates the detailed architectural design of the system. In order to implement the abstraction layers, the Hybrid System implementation consists of various modules such as Query and Publishing, Expeditor, Access, Storage, Filtering and Information Resource Manager and Sequencer. We discuss the abstraction layers of the system within the context of the system's modular structure in the following sections.

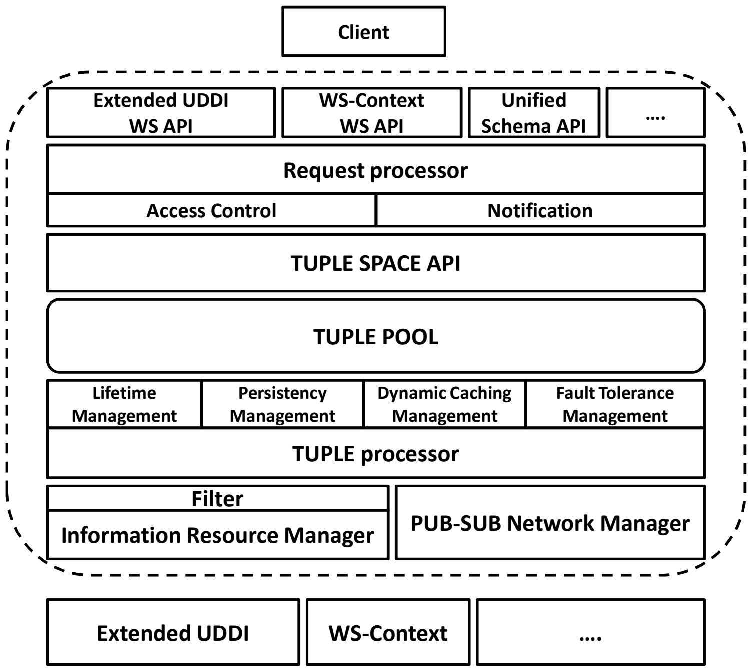


Figure 5 The Architectural Design for the Hybrid Grid Information Service

3.4.1 Query and Publishing Module

The Query and Publishing module implements the following abstraction layers: 1) The uniform access interface layer, 2) The request-processing layer, and 3) The access control and notification layers. 1) 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 5, 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. See Section 4.4 for details.). The access interface can import more XML API, as the new information services are integrated with the system. 2) The request-processing layer is responsible for extracting incoming requests and process operations on the Hybrid Service. 3) 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 abstraction layer is responsible for enforcing controlled access to the Hybrid Grid Information Service.

3.4.2 Expeditor Module

The Expeditor module implements the following abstraction layers: 1) Tuple Spaces Access layer, 2) Tuple Pool, 3) Tuple-processing and various capability layers. 1) TupleSpaces Access API allows access to in-memory storage. This API supports all query/publish operations that can take place on the Tuple Pool., 2) The Tuple Pool implements a lightweight implementation of JavaSpaces Specification (see Section 2.3 for details) and is a generalized in-memory storage mechanism. It enables mutually exclusive access and associative lookup to shared data, 3) The tuple processor is being used to provide various capabilities. 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.

3.4.3 Filter and Resource Manager Modules

In this thesis, to facilitate testing of the Hybrid Service federation capability, we introduce a Unified Schema and its Query/Publish XML API to provide federation of information in Grid Information Services. To illustrate the federation capability, the Unified Schema is constructed by integrating three information service schemas: Extended UDDI, WS-Context and Glue (see 4.4.1 for more details). We discuss the abstract data models and communication protocols of the Unified Schema in Section 4.4 in detail. This approach is introduced to provide a common communication platform among the existing information services.

The Filtering module is implemented to support the federation capability of the Hybrid System. It implements the filtering layer, which provides filtering capability based on the user defined mapping rules to provide transformations between instances of the Unified Schema and local information service schemas such as WS-Context Schema.

The Information Resource Manager component implements the Information Resource Management layer. This layer is responsible for managing low-level information service implementations. It provides decoupling between the Hybrid Service and sub-systems.

3.4.4 Sequencer Module

The Sequencer module ensures that an order is imposed on actions/events that take place in a session. It is used to label each metadata, which will be stored in the system, with synchronized timestamps.

3.4.5 Storage and Access Modules

Both Storage and Access Modules implements the Pub-Sub Network Management abstraction layer. The Storage module mainly handles with replica placement, dynamic replication and consistency enforcement. The Access module handles with request distribution. It deals with the problem of redirecting a client request to the appropriate replica server. Both Access and Storage modules utilize topic based publish-subscribe paradigm to interact with the other nodes in the system.

3.5 Supported Information Service Specifications

The Hybrid Grid Information Service presents an architecture, which extends the capabilities of customized implementations of information service specifications. This add-on capability enables <u>unification</u> and <u>federation</u> of information in Grid Information Services.

To facilitate testing of the <u>unification</u> capability, we provided implementations of the two specifications: the WS-Context Specification and the UDDI Specification. Thus, the prototype implementation of the Hybrid Service supports the WS-Context and Extended UDDI XML Metadata Services. The WS-Context XML Metadata Service manages session-related, interaction-dependent metadata associated to Grid/Web Services. The extended UDDI XML Metadata Service is an implementation of extended version of existing UDDI Specification. It manages interaction-independent, rarely changing metadata associated to Grid/Web Services.

To facilitate the testing of the federation capability, we introduced a Unified Schema Specification. With the Unified Schema Specification, we integrate different information service data models under one Unified Schema and provide an XML API to publish/inquire Unified Schema metadata instances.

3.5.1 WS-Context Specification

Section 2.1.2 discussed the WS-Context Specification and its limitations. The WS-Context Specification defines session related, interaction-dependent metadata. This thesis implements the WS-Context XML Metadata Service, which presents a schema and XML API for the Context Manager component of the WS-Context Specification. The WS-Context Service implementation 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 semantics of the extended version of the WS-Context Specification are discussed in Section 4.2, while the prototype implementation is discussed in Section 5.7

3.5.2 Extended UDDI Specification

Section 2.1.1.4 discussed the limitations of existing UDDI Specifications. This thesis presents an extended UDDI Specification addressing these limitations and provides an implementation: the extended UDDI XML Metadata Service. The extended UDDI Service provides a metadata-oriented storage capability. 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 application use domain discussed in Section 1.3.2.2. 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. The semantics of extended UDDI Specification are discussed in Sections 4.3, while the prototype implementation is discussed in Section 5.8.

3.5.3 Unified Schema Specification

The Hybrid Service supports a federation capability where different Grid Information Services can be integrated in metadata instances. This thesis introduces a Unified Schema, which integrates different information service schemas and provides Query/Publish XML API that can be carried out on the metadata instances of the Unified Schema. This enables the Hybrid Service clients to publish/query metadata describing all aspects (interaction-independent and interaction-dependent) of Grid/Web Services. The semantics of the Unified Schema is discussed in Section 4.4.

3.6 Summary

This chapter presented the architectural design of the Hybrid Grid Information Service. (The detailed XML API for the Hybrid Service is given in the Appendix A). To achieve this goal, first, an overview of the system was given. Then, its network communication model was discussed. Next, the assumptions on which the system is built were presented followed by brief descriptions of its modular components. Finally, the information service specifications, which are supported by the Hybrid System, were introduced. Chapter 4 discusses the semantics of the proposed system, while Chapter 5 discusses the prototype implementation.

Chapter 4

Abstract Data Models

Geographical Information Systems provide very useful problems in supporting "virtual organizations" and their associated information systems. These systems are comprised of various archival data services (Web Feature Services), data sources (Web-enabled sensors), and map generating services. All of these services are metadata-rich, as each of them must describe their capabilities (What sorts of geospatial features do they provide? What geographic bounding boxes do they support?). Organizations like the Open Geospatial Consortium define these metadata standards. These services must typically be assembled into short-term service collections that, together with code execution services, are combined into a meta-application (i.e. a workflow). Thus, we see that we have both stateless and stateful (transient) metadata. To address the problems of metadata management in Geographical Information Systems-like application use domains (see Section 1.3.2.1), we have investigated semantics for Information Services that can provide uniform access interface to stateless and stateful information associated to services.

4.1 Overview

We designed and implemented a novel architecture [15-17, 25, 108] for a Hybrid Grid Information Service supporting handling and discovery of not only quasi-static, stateless metadata, but also session related metadata. The Hybrid Service runs as an add-on architecture above existing information services. It provides unification and federation of information in Grid Information Services. The Hybrid Service provides a uniform access interface that allows users to publish/inquire metadata instances by utilizing different information service XML APIs.

Firstly, to achieve the unification capability with the Hybrid Service, we built two information services: WS-Context XML Metadata Service and Extended UDDI XML Metadata Service. To implement these services, we utilized the two WS-I compatible Specifications: Web Services Context (WS-Context) [82] and Universal Description, Discovery, and Integration (UDDI) [8]. The WS-Context Service is implemented based on WS-Context Specification. It is an implementation of the Context Manager component of the WS-Context Specification. We designed a schema and XML API set to provide search/access/storage interface for the session-related metadata. The Extended UDDI Service is implemented based on extended UDDI Specification which we designed by expanding on out-of-box UDDI Specifications. We designed extensions to out-of-box UDDI Data Structure and UDDI XML API set to be able to associate both prescriptive and descriptive metadata with service entries.

Secondly, to achieve the federation capability with the Hybrid Service, we built a Unified Schema Specification. This Specification introduces a Unified Schema, which integrates different information service data models. For schema integration, we consider three Information Service Schemas: Extended UDDI, WS-Context and Glue [111]. To be able publish/inquire metadata instances of the Unified Schema; we also introduce an XML API set.

Thirdly, to achieve a uniform access interface with the Hybrid Service, we provided a Hybrid Schema, which describes how to interact with the system in a uniform way, and XML API that would allow the users to publish/inquire metadata instances utilizing XML APIs of aforementioned specifications.

In this section, we discuss the semantics of the supported specifications and Hybrid Service in detail. We have identified the following base elements of the semantics: a) information model (data semantics), b) XML programming interface (semantics for publish and inquiry XML API).

4.2 The WS-Context Specification Semantics

Chapter 2 introduced varying ways of managing context associated with services. Section 2.1.2 described methodologies for managing interaction-dependent context. Among these methodologies, we find WS-Context promising to tackle the problem of managing distributed session state. Unlike the point-to-point approaches explained in Section 2.1.2, 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.

4.2.1 WS-Context Schema

We introduced an information model comprised of following entities: sessionEntity, sessionService and context. Figure 6 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.

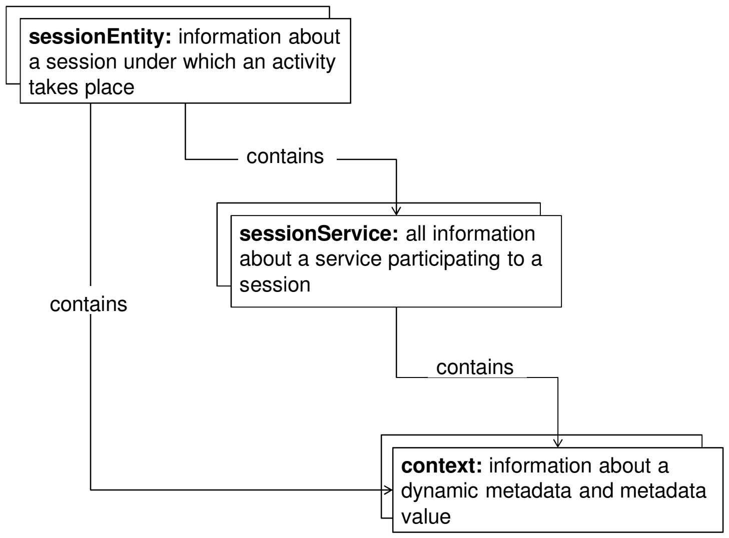

sessionEntity: information about a session under which an activity takes place
contains
sessionService: all information about a service participating to a session
contains
contains
context: information about a dynamic metadata and metadata value

Figure 6 WS-Context Service Schema

4.2.1.1 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 7. 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.

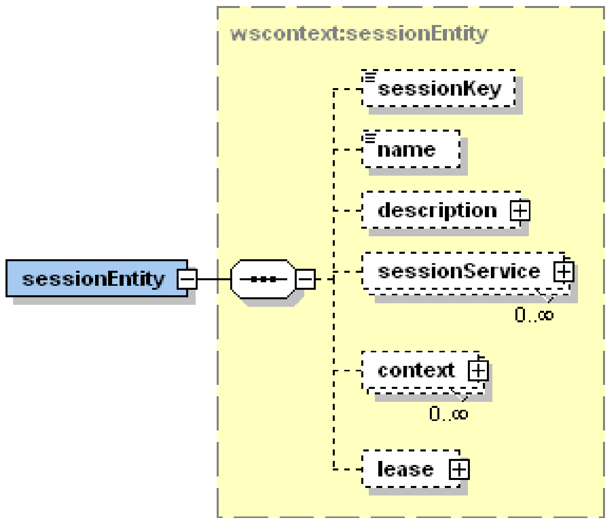


Figure 7 Structure diagram for sessionEntity

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. (See 4.2.1.3 for context entity structure). 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 (see 4.2.1.2 below for session service entity structure). A lease structure describes a period of time during which a sessionEntity or serviceService or a context entity instances can be discoverable.

4.2.1.2 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 8. 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.

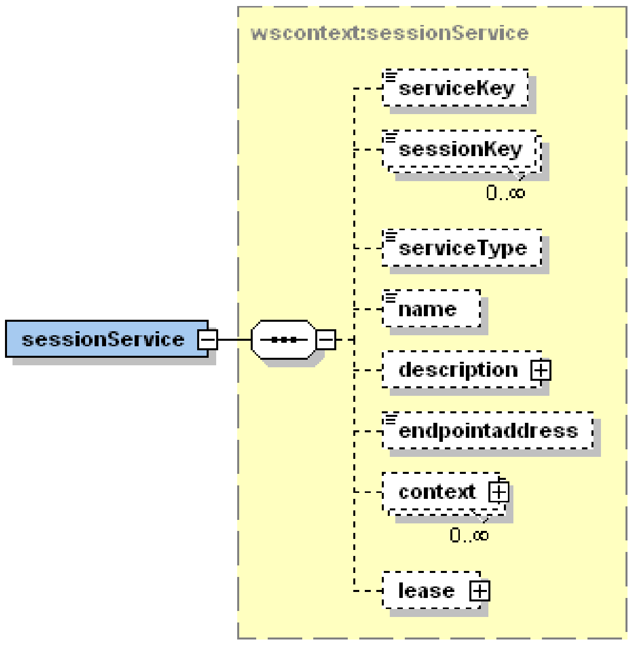


Figure 8 Structure diagram for sessionService

4.2.1.3 Context entity structure

A context entity describes dynamically generated metadata. The structure diagram for context entity is illustrated in Figure 9. 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.

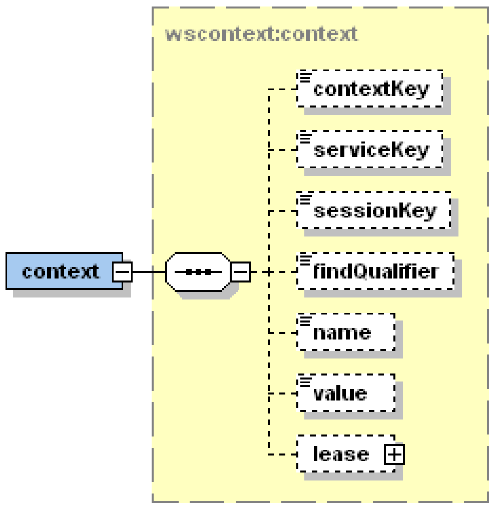


Figure 9 Structure diagram for context entity

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.

4.2.2 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. Appendix A.1 gives the detailed descriptions about the syntax, arguments, and return values of the WS-Context XML API Sets. Table 2 gives the list of available XML API, which we introduce with the WS-Context Service.

Function	**Category**	**Information Service**
Save_context	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_session		
Save_sessionService		
Delete_context		
Delete_session		
Delete_sessionService		
Get_contextDetail	Inquiry Functions	
Get_sessionDetail		
Get_sessionServiceDetail		
Find_context		
Find_session		
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

The Publish XML API is used to publish metadata instances belonging to different entities of the WS-Context Schema. The Inquiry XML API is used to pose inquiries and to retrieve metadata from service. The Proprietary XML API is implemented to provide find/add/modify/delete operations on the publisher list, i.e., authorized users of the system. The Security XML API is used enable authenticated access to the service. Here, we adapt semantics for both the proprietary XML API and security XML API from existing UDDI Specifications.

4.2.3 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 complete example of a context is given in the Appendix B.1. 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 Extended UDDI Specification Semantics

We have designed extensions to the out-of-box UDDI Data Structure (described in [8]) 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 [75] and UDDIe [71] 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.

4.3.1 Extended UDDI Schema

We introduced an extended UDDI data model 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.

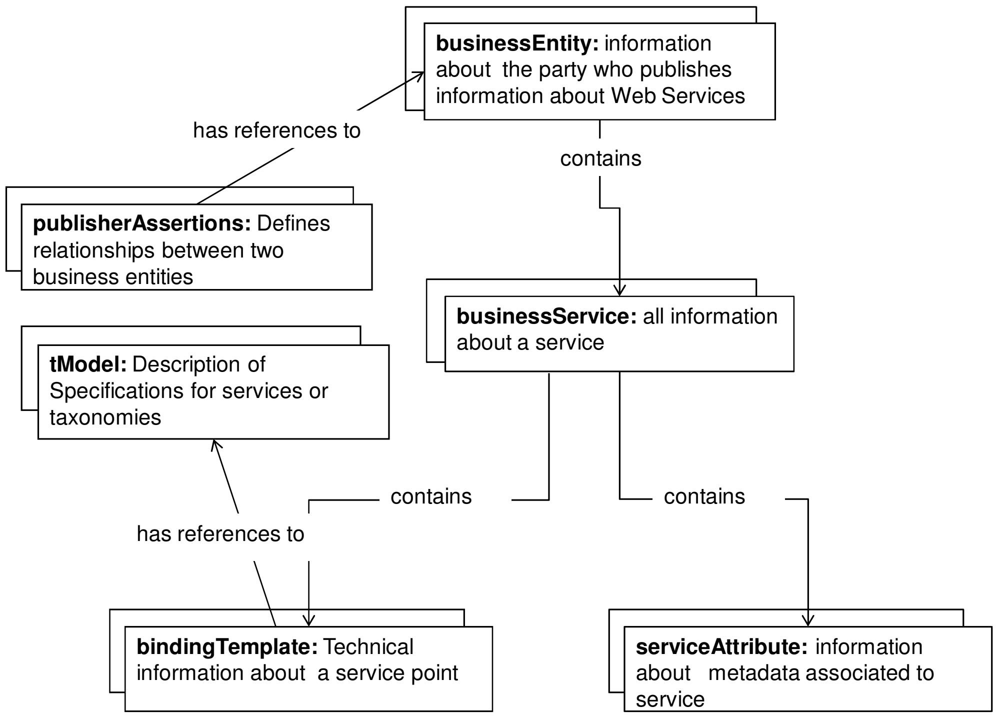

businessEntity: information about the party who publishes information about Web Services
has references to
contains
publisherAssertions: Defines relationships between two business entities
businessService: all information about a service
tModel: Description of Specifications for services or taxonomies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

Figure 10 Extended UDDI Service Schema

4.3.1.1 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 [8]. 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 11)

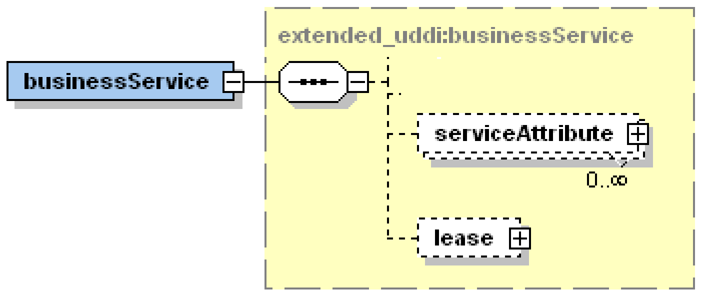


Figure 11 Partial structure diagram for businessService entity

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.

4.3.1.2 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 12. Each service attribute corresponds to a piece of metadata, and it is simply expressed with (name, value) pairs. Apart from similar approaches [71, 75], 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.

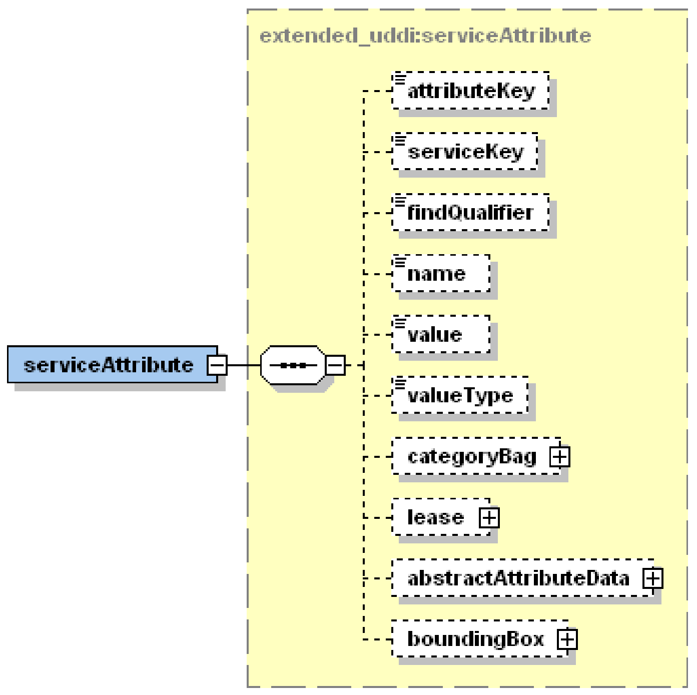


Figure 12 Structure diagram for serviceAttribute

4.3.2 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. Table 3 gives the list of additional XML API that we introduce with the Extended UDDI Service.

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

Table 3 The Publish/Inquiry XML API for the extended UDDI Service. The extended UDDI XML API is introduced an extension to existing UDDI Specification XML API Sets.

The Publish XML API is used to publish metadata instances belonging to different entities of the extended UDDI Schema. The Inquiry XML API is used to pose inquiries and to retrieve metadata from the Extended UDDI Information Service. More detailed information about syntax, arguments, and return values of the programming API sets are given in Appendix A.2.

4.3.3 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.

4.4 The Unified Schema Specification Semantics

The Hybrid Grid Information Service provides a federation capability that enables integration of different information services in metadata instances. With this capability, our aim is to introduce an architecture, which would support an integrated schema by utilizing expressiveness power of different information service schemas. To facilitate the testing of this capability, we created a Unified Schema that would integrate different information service schemas. We consider the schemas ExtendedUDDI, Glue and WS-Context as a motivating example to create the Unified Schema. The Extended UDDI and the WS-Context Schemas are described in Sections 4.2 and 4.3 respectively in detail. We discuss the Glue Schema Specification in the next section. We discuss the methodology for integrating these Schemas in Section 4.4.2.

4.4.1 The Glue Schema Specification

The Grid Laboratory Uniform Environment (Glue) Schema [111] 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 Information Model of the Glue Schema is given in [111]. 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.

4.4.2 The Schema Integration

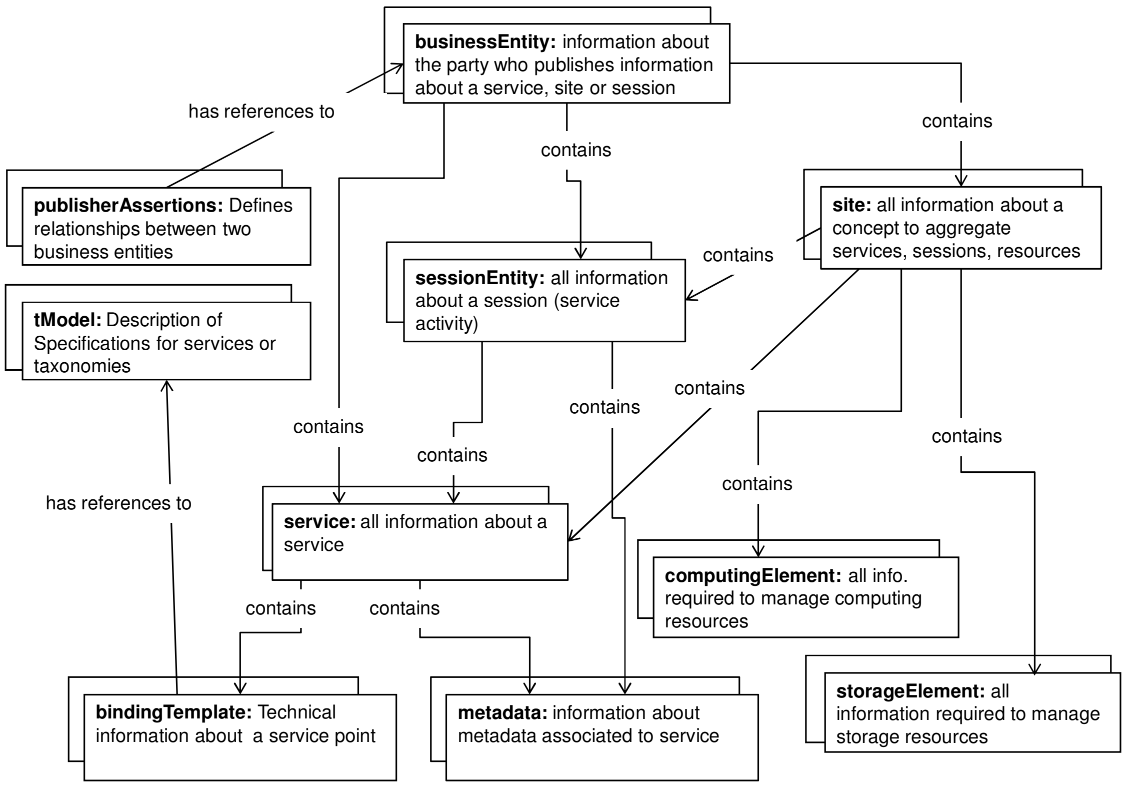
Schema integration is an activity of providing a unified representation of multiple data models [112]. The schema integration consists of two core steps: schema matching [112] and schema merging [113]. The schema matching step identifies mapping between the similar entities of schemas. Matching between 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 aggragates 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 13 for illustration) that integrates the Extended UDDI, WS-Context and Glue Schemas.

4.4.3 The Unified Schema

We built a Unified Service Schema integrating the extended UDDI, the WS-Context and the Glue Schemas [111] by following the steps described in the previous section. 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. It is comprised of the following entities: businessEntity, sessionEntity, site, service, computingElement, storageElement, bindingTemplate, metadata, tModel, publisherAssertions. Figure 13 illustrates the information model for the Unified Schema. 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.



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
contains
sessionEntity: all information about a session (service activity)
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
storageElement: all information required to manage storage resources
bindingTemplate: Technical information about a service point
metadata: information about metadata associated to service

Figure 13 Unified Schema

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.

4.4.4 The Unified Schema XML API

We introduce a Query/Publish XML API that can be carried out on the instances 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 4 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.

The Publish XML API is used to publish metadata instances belonging to different entities of the Unified Schema. The Inquiry XML API is used to pose inquiries and to retrieve metadata instances of the Unified Schema. More detailed information about syntax, arguments, and return values of the programming API sets are given in Appendix A.3.

4.4.5 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 complete example of a context is given in the Appendix B.3. 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 Uniform Access 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 Schema. The Hybrid Schema defines the generic access interface. The Specification Metadata Schema defines the necessary information required by the system to support a specification. We discuss the semantics of the uniform access interface and the specification metadata in the following sections.

4.5.1 The Hybrid Service Schema

The Hybrid Service presents an XML Schema, called the Hybrid Schema, to enable uniform access to the system. The Hybrid Schema defines publish and inquiry XML API which allows clients/providers to send specification-based publish/query requests (such as WS-Context's "save_context" request) to the system. To illustrate the Hybrid Service access interface, we only discuss the "save_schemaEntity" element (see Figure 14), which is used to publish metadata instances into the Hybrid Service. More detailed information about syntax, arguments, and return values of the XML API sets are given in Appendix A.4.

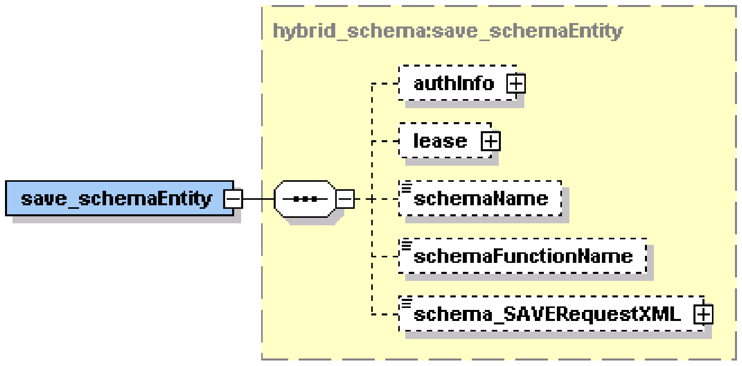


Figure 14 Hybrid Service XML Schema for Hybrid Service metadata publish function

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 Specification Metadata Schema

The Specification Metadata 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 15. The Hybrid System requires an XML metadata document, which is generated based on the Specification Metadata Schema, for each information service specification supported by the system. The specification metadata file enables the Hybrid System to know how to process instances of a specification XML API.

The specification metadata includes name, description, and version of the specification under consideration. These are the descriptive information to help the Hybrid Service to identify the local information service schema under consideration.

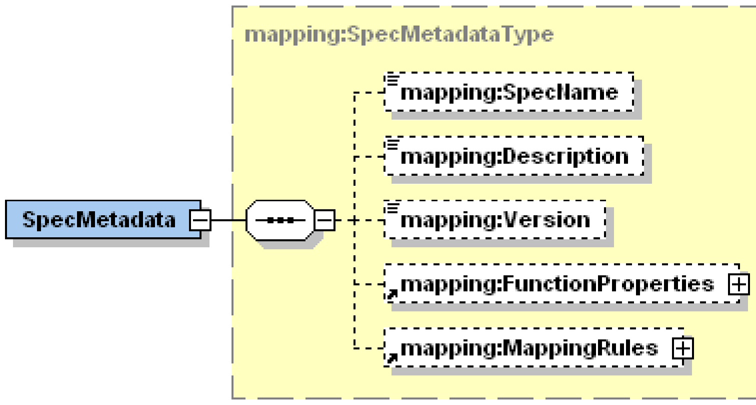


Figure 15 Structure diagram for Specification Metadata Schema: This metadata file defines all required information necessary to support a new information service

The FunctionProperties 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 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.

4.5.3 Using the Hybrid Service Access Interface

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.

4.6 Summary

This chapter presented the semantics of the information services presented in this thesis. First, it presented the semantics of the WS-Context Specifications. Second, it presented the semantics of the extended UDDI Specifications. Third, it presented the semantics of the Unified Schema Specification. Finally, we introduced the semantics for the Hybrid Grid Information Service Uniform Access Interface.

Chapter 5

Prototype Implementation

This chapter presents implementation details of a prototype of the aforementioned system architecture. The purpose of the prototype is to validate the system architecture of Chapter 3 and abstract data models of Chapter 4. This prototype is implemented by utilizing following technologies and open-source research projects: a) Java 2 SDK, Standard Edition with version 1.5 [114], b) Apache Axis Web Service Development Platform with version 2 [115], c) NaradaBrokering Messaging Infrastructure with version 1.1.6 [116], and d) Apache JUDDI project, an open-source java implementation of the UDDI Specification [117]. Our implementation is also open-source and available from [118]. The implementation of the proposed architecture can be structured under three distinct systems: the Hybrid Grid Information Service, the WS-Context Service and the extended UDDI Service.

5.1 Hybrid Grid Information Service

We implemented a fault tolerant and high performance Hybrid Grid Information Service. As described in Chapter 3, each Hybrid Grid Information Service consists of various modules such as Query and Publishing, Expeditor, Filter and Resource Manager, Sequencer, Access and Storage. 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 Module provides 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.

In our design, we focus on three fundamental issues of designing a replica hosting system: replica-content placement, request routing and consistency enforcement. Replica content placement has to do with creating a set of duplicated data replicas across the nodes of a distributed system. Request routing has to do with redirecting a client request to the most appropriate replica server. Consistency enforcement deals with ensuring data coherency across replicas in the system.

In the modular structure of the system architecture, the Storage module covers the replica-content placement and consistency enforcement issues, while the Access module implements the request routing. The communication between the nodes is done by using a multi-publisher, multicast communication mechanism (see Section 3.2). The system utilizes the NaradaBrokering [89, 110] software, an open-source, publish-subscribe based messaging infrastructure, to provide such communication. A node is connected to another via NaradaBrokering link, which in turn creates an overlay network that connects the Hybrid Service nodes. We discuss the execution flow of the system and the abstraction layers of the implementation in the following section.

5.1.1 Execution Logic Flow

Figure 16 illustrates the execution logic flow for the Hybrid Grid Information Service. Firstly, the proposed system presents a uniform access layer. This abstraction layer supports one to many communication protocol of different information services.

Secondly, the system presents a request-processing layer. 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 metadata files (see Section 4.5.2). For each supported schema, there is a specification-mapping metadata 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.

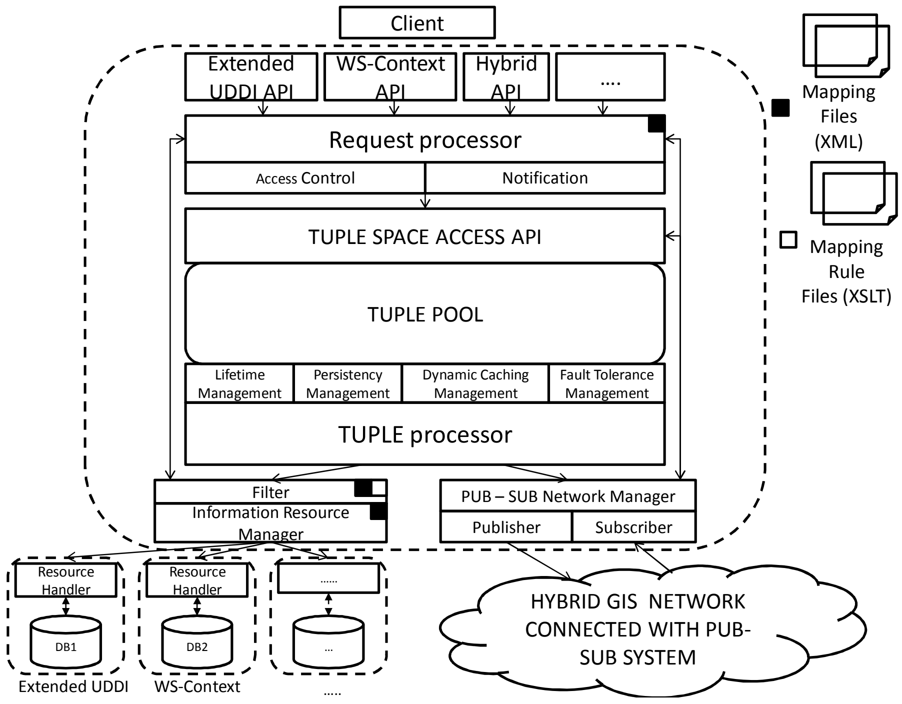


Figure 16 Execution Logic Flow for the Hybrid Grid Information Service. This figure illustrates the execution flow of the Hybrid Grid Information Service from top-to-bottom. Each rectangle shape identifies a layer of the system with particular purpose. The square-black color shapes indicate that the 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.

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.

Thirdly, once the request is extracted and processed, the system presents abstraction layers for some capabilities such as access control management and notification. First capability is the Access Control Management. This capability layer is intented to provide access controlling for metadata accesses. As the main 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.

Fourthly, 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.

Fifthly, 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.

The Hybrid Service supports a federation capability to address the problem of providing integrated access to heterogenous metadata. To facilitate the testing of this capability, this thesis introduces a Unified Schema by integrating different information service schemas (see Section 4.4). 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 a 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.

Sixthly, 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 sevice 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 which enables interaction with the Hybrid Service.

Seventhly, 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 a Publisher and a Subscriber in order to provide communication among the instances of the Hybrid Services.

5.2 Query and Publishing module

The Query and Publishing module is responsible for implementing a uniform access interface for the Hybrid Grid Information Service (see Chapter 4 for detailed discussion on the semantics of XML API sets supported by the system). This module implements the Request Processing, Access Control and Notification Management abstraction layers explained in the previous section.

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 the context information may not be open to anyone, there is a need for an information security mechanism. We leave out the investigation and implementation of this mechanism for the decentralized Hybrid Service as a future study. We define an Access Control abstraction layer that will be responsible for providing controlled access to metadata instances. We must note that to facilitate testing of the centralized Hybrid Service in various application use domains (see Section 1.3.2), we implemented a simple 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 to meet the requirements of application use domains discussed in Section 1.3.2. 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. This methodology is especially become useful for the application use domains where the consistency is important. Based on this scheme, state changes are propagated to the interested clients by the primary-copy holding service whenever an update occurs. (see Section 5.6.8.2 for more details on update propagation). We use the aforementioned NaradaBrokering software (see Section 2.2) as the messaging infrastructure and its libraries to implement subscriber and publisher components.

5.3 Expeditor module

This module implements the following abstraction layers: 1) Tuple Spaces Access layer, 2) Tuple Pool, and 3) Tuple Processing layers. The Tuple Spaces Access layer provides an access interface on the Tuple Pool, which is a generalized in-memory storage mechanism. Here, we built the in-memory storage based on the TupleSpaces paradigm [94] (see Section 3.4.2 for detailed discussion). 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. (In this study, we adopt the dynamic replication methodology, introduced by Rabinovich et al, which will be discussed in Section 5.6.7 in length).

5.4 Filter and Resource Manager Modules

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, illustrated in Figure 17, handles with management of local information service implementations such as the extended UDDI. The Resource Manager 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 (see Section 4.5.2) that belongs the local information service under consideration.

On receipt of a request, the Information Resource Manager checks with the corresponding mapping file and obtains 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 files are provided.

The Resource Handler implements the sub-information system functionalities. 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

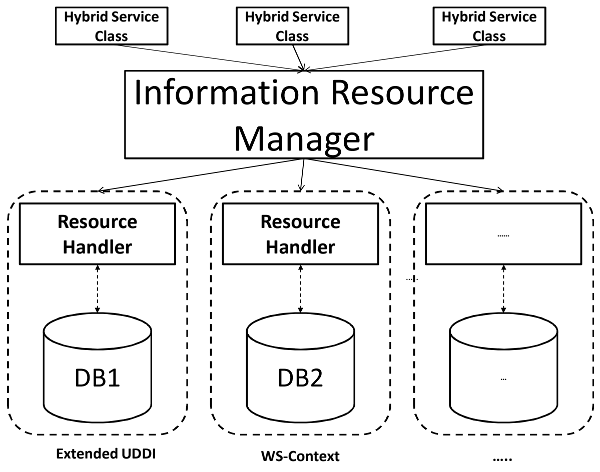


Figure 17 We implemented an Information Resource Manager, which separates specification-implementations from the implementation of the Hybrid Service.

The Resource Manager component 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.

5.5 Sequencer module

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 [119] implemented by NaradaBrokering software (see Section 2.2). This service achieves synchronized timestamps by synchronizing the machine clocks with atomic timeservers available across the globe. The Sequencer module is also used to generate unique identifiers to assign system-generated keys to newly stored dynamic metadata. This is succeeded by utilizing a java UUID generator (JUG) [107], which is an open-source, free, java implementation of the IETF UUID Specification [120]. The Sequencer module interacts with the Expeditor module to assign unique identifiers or timestamps to the newly stored or updated contexts.

5.6 Access and Storage modules

Distribution of client requests is the problem of redirecting a client request to the appropriate replica server. In the modular structure of our design, the Access module supports the request distribution by publishing messages to topics in NaradaBrokering software multicast system.

The Storage module handles with replica-content placement and consistency enforcement. It interacts with data-systems that can store a replica and provides replication. It also ensures data consistency among replicas. It interacts with the Expeditor module to access local data maintained in Tuple Pool.

Replica placement issue consists of two sub-problems: replica server placement and replica content placement [102]. The former issue deals with the problem of finding suitable locations for replica servers, while the latter issue handles with selecting replica servers that should host a data. In this research, we study the latter problem, which concerns with the selection of replica servers that must hold the data under consideration.

A major design issue for distributed data-systems is to decide where, when, and by whom copies of a data are to be placed [121]. Tanenbaum discusses three different kinds of copies of a data in [103]: permanent, server-initiated, and client-initiated. Permanent replicas can be considered as an initial replica-set comprised of minimum required number of copies of a data. This type of replica is used to provide a certain degree of fault-tolerance. Server-initiated replicas are considered temporary and created by a data-system in a dynamic fashion to improve the performance. For example, server-initiated replicas can be created to handle sudden and big number of requests coming from a location far away from the server. The Hybrid System is implemented to support both permanent and server-initiated types of replica. Server-initiated replication is introduced to enhance the performance in terms of minimizing the latency. We utilize the dynamic replication methodology introduced by Rabinovich et al [104] to control the server-initiated replicas. The dynamic replication technique allows us to gradually decrease or increase the popularity of server-initiated replicas depending on the changing client demands (Section 5.6.7 will discuss the dynamic replication in length). Permanent copies of a data are important to at least keep the minimum required number of replicas for the same data. The client-initiated replicas are the copies of a data that are just requested and temporarily stored at the client applications. The management of client-initiated replicas of a data belongs to client applications, thus it is not in scope of the Hybrid System server side implementation.

5.6.1 Tunable Parameters

In order to provide replica-content placement, access distribution, dynamic replication and consistency enforcement in replica hosting system, the following tunable parameters are used: *backup-time-interval*, *dynamic-replication-time-interval*, *minimum-fault-tolerance-watermark, maximum-server-load-watermark, timeout-period, deletion-threshold* and *replication-threshold.*

backup-time-interval: In order to provide persistency, metadata instances in the Tuple Pool are backed-up into a persistent storage (such as MySQL database) with certain time intervals (*backup-time-interval*). There is a trade-off in choosing the value for *backup-time-interval*. If the *backup-time-interval* is chosen to be too small, then the system performance will be lower. If this time interval is too big, then the system will be less persistent. (See Section 6.2 for our investigation on *backup-time-interval.*)

dynamic-replication-time-interval: In order to provide dynamic replication, metadata instances in the Tuple Pool are replicated in replica-hosting environment in a dynamic fashion within certain time intervals (*dynamic-replication-time-interval*). The trade-off in choosing the value for *dynamic-replication-time-interval* is similar to the one for *backup-time-interval*. If the *dynamic-replication-time-interval* is chosen to be too small, then the system performance will be affected. If this time interval is too big, then the system will not adapt well to changes in client demands such as sudden bursts of request that come in from an unexpected location. (Rabinovich et al introduced an extensive study on choosing values for the dynamic-replication tunable parameters. In our investigation, we chose the simulation parameters relying on their study in [104].)

minimum-fault-tolerance-watermark: To provide a certain level of fault-tolerance, we use a *minimum-fault-tolerance-watermark* indicating minimum required degree of replication. The trade-off in choosing the value for *minimum-fault-tolerance-watermark* is the following. If the value is chosen to be high, then the time and system resources required completing replica-content placement and keeping these replicas up-to-date would be high. If the value is chosen to be too small, then the degree of replication (fault-tolerance level) will be low. (See Section 6.6 for our investigation on *fault-tolerance levels.*)

maximum-server-load-watermark*:* To avoid overloading a single Hybrid Service, we use a tunable parameter *maximum-server-load-watermark* and a decision metric instantaneous-server-load. For an example, a given Hybrid Service node can process an incoming storage request, if the instantaneous-server-load would not exceed *maximum-server-load-watermark*, which is predefined in the configurations file. Otherwise, this request should be forwarded to another server. As for the instantaneous-server-load metric, we use the message rate information as an indication of the load on the system at a given time interval. The trade-off in choosing the value of *maximum-server-load-watermark* is as in the following. If this value is chosen to be too high, then the Hybrid Service performance will decrease. If the value is chosen to be too low, then the Hybrid Service will be running under its potentials. (See Section 6.3 for our investigation on message rate scalability.)

timeout-period: The tunable *timeout-period* value indicates the amount of time that a Hybrid Service node is willing to wait to receive response messages. The trade-off in choosing this number is the following. If the *timeout-period* is too small, the initiator of a request will not wait enough for the context access responses coming from a multicast group. For example, if there are two replica servers, one in U.S. and the other in Australia, the query initiator located in U.S. may miss the result coming from the node located in Australia with a small *timeout-period*. If the *timeout-period* is too big, then the query initiator may have to wait for a long time unnecessarily for some information that does not exist in the replica-hosting environment.

deletion-threshold: If a temporary-copy (server-initiated) of a context is in low demand and its demand count is below *deletion-threshold*, then this temporary copy needs to be deleted. The *deletion-threshold* determines the rate for migration and replication occurring in the system. If a *deletion-threshold* is selected too low, the system will create more temporary copies, which will lead into high number of message exchanges in the system. If a deletion-threshold is too high, the system will keep low-demand temporary copies of a context unnecessarily. In our investigation, we chose the *deletion-threshold* value based on the study introduced in [104].

replication-threshold: If a context is in high demand and its demand count is above a *replication-threshold*, then the context is replicated as a temporary-copy. If the *replication-threshold* is selected to be too high, then the system will not adapt well to high number of client demands. If the *replication-threshold* is too low, the system will try to create temporary replicas at every remote replica where small number of requests comes in. This may cause unnecessary consumption of system resources. (Rabinovich et al [104] discusses the dependency between replication and deletion thresholds that in turn indicates that the value of *replication-threshold* must be selected above *deletion-threshold.* In our investigation, we chose the *replication-threshold* value based on the study introduced in [104].)

5.6.2 Decision Metrics

The Hybrid Service uses some measurements to decide on replica-content placements. Our replica-server selection policy takes both server load and proximity decision metrics into account when making replica-content placement decisions. The server load metric is a decision metric, which may be represented with multiple factors. We used the following two factors: a) topical information (i.e. number of unique topics, which the Hybrid Service subscribe to) and b) message rate (i.e. number of messages, issued by end-users, within a unit of time). If the number of topics, which a Hybrid Service subscribes to, is high, it is likely that the Hybrid Service will receive high number of access/update messages. If the message rate on a given Hybrid Service is increased, its performance will start dropping down. Therefore, we take into consideration the topical and message rate information as server load metrics. Each node can estimate its own server load based on these two factors. Server load is periodically recorded and it reflects the average load of a Hybrid Service at a given time interval. Note that, each nodes keeps decision metrics information about other nodes in the system. The server load information is obtained periodically by sending a Server-Information Request message to other available network nodes in the system. The proximity metric is the decision metric, which is used to indicate the distance in network space between Hybrid Service instances. The proximity metric information is obtained periodically by sending ping requests

(Server-Information Requests) to the available network nodes in the system through publish-subscribe system topics. The latency in the ping request gives the proximity information between the two Hybrid Service instances.

5.6.3 Control Data Structures

The system keeps following control data structures in decision-making process of replica-content placement, dynamic replication and consistency enforcement: *is-context-removable-flag, access-demanding-server-info, access-request-count, replica-server-info-map* and *version-number.*

is-context-removable-flag*:* In order to ensure that there exists certain number of permanent replica-set of a given context, *is-context-removable-flag* control variable is used. This variable is used to differentiate permanent and server-initiated replicas and contained within the Tuple object. If this variable is true, then a context replica is considered as server-initiated and can be deleted by the dynamic replication algorithm (see Section 5.6.7) unless there are enough clients demanding it. If it is false, then this copy is considered as permanent and cannot be deleted for fault-tolerance reasons.

access-demanding-server-info*:* Each Hybrid Service node, let us say *s*, keeps track of certain information, *access-demanding-server-info*, about access requests aimed at a context *x* where ($x \mid x$ is a context hosted by *s*). This information is used by dynamic replication algorithm to find those replica servers, which demanded the context under investigation, the most. The *access-demanding-server-info* includes the number of access request counts per context and the hostname (e.g. IP address) of the replica server where access requests come from.

access-request-count: The accumulated value of all individual access requests (made by different nodes in the system) gives the total number of demand, *access-request-count*, on a particular context. This value is used by the dynamic replication algorithm to control dynamic migration and replication of a given context replica.

replica-server-info-map: Each Hybrid Service node, let us say *s*, keeps track of certain information, *replica-server-info*, about other available servers in the system. This information is used by replica-content placement algorithm to find best replica servers, which are capable of storing the permanent-copies of a context under investigation. The *replica-server-info* includes the hostname (e.g. IP address) of the replica other server, their proximity and server load information.

version-number: The *version-number* is the synchronized timestamp of the last update.

5.6.4 Network Messages

The communication between Hybrid Service nodes happen via message exchanges. There are various messages designed to enable communication between the nodes of the network to enable replica-content placement, access distribution, dynamic replication and consistency-enforcement. These messages are Server-Information Request and Response, Context Access Request and Response, Context Storage Request and Response, Primary-Copy Selection Request and Response, Primary-Copy Notification, Context Update Request and Propagation messages. We discuss the purpose and dissemination methodology of these messages in the following sections.

5.6.4.1 Server-Information Request and Response messages

A Hybrid Service node advertises its existence when it first joins the network with a message, the Server-Information Request. The purpose of the Server-Information Request message is two-fold. First purpose is to inform other servers about a newly joining server. Second purpose is to refresh the replica-server-information data structure with the updated information (such as proximity and server load information) every so often. This message is broadcasted through publicly known topic to every other available network nodes. The proximity between the initiator and the individual network nodes is calculated based on the elapse-time between sending off the Server-Information Request and receiving the Server-Information Response message. The Service-Information Response message is sent back by unicast over a unique topic (IP_Address) to the initiator. This message also contains the server load information of the responding network node.

5.6.4.2 Context Access Request and Response messages

A Hybrid Service node advertises the need for context access with the Context Access Request to the system. The purpose of the Context Access Request message is to ask those servers, holding the context under demand, for query handling. This message is disseminated to only those nodes holding the context under consideration. This is done by multicasting the message through the unique topic corresponding to the metadata. (Note that we use UUID of the metadata as topic). By listening to this topic, each node, holding the context under consideration, receives a Context Access Request message, which in turn includes the context query under consideration. On receipt of a Context Access Request message, each Hybrid Service sends a Context Access Response message, which contains the context under demand, to the initiator. This message is sent out by unicast directly to the initiator over a unique topic. (Note that we use IP address of the initiator as topic to send responses via unicast back to the initiator). By listening to this topic, the initiator receives the response messages from nodes that answered the access request.

5.6.4.3 Context Storage Request and Response messages

A Hybrid Service node advertises the need for storage with a request message, the Context Storage Request. The purpose of the Context Storage Request message is two-fold. First purpose is to assign handling of the storage operation to those Hybrid Service nodes that are selected based on the replica-server selection policy. Note that this message is used in replica-content placement process, which is discussed in Section 5.6.6. The second purpose is to ask another Hybrid Service node to replicate or take over maintaining a context to enhance the overall system performance. Note that with this message, the system is able to relocate/replicate contexts in the proximity of demanding clients. It is used in dynamic replication process, which is discussed in Section 5.6.7 and enables relocation/replication of contexts due to changing client demands. The Context Storage Request message is unicast over a unique topic to the selected replica server(s). By listening to its unique topic (IP_Address), each existing node receives a Context Storage Request message, which in turn includes the context under consideration. On receipt of a Context Storage Request message, a Hybrid Service node stores the context and sends a Context Storage Response message to the initiator. The Hybrid Service stores the context either as a permanent-copy or server-initiated (temporary) copy based on whether the context is being created for fault-tolerance reasons or performance reasons. The purpose of the response message is to inform the initiator that the answering node hosts the context under consideration. This message is also sent out by unicast directly to the initiator over a unique topic (IP_Address). By listening to this topic, the initiator receives response messages from the nodes that handled the storage request.

5.6.4.4 Primary-Copy Selection Request and Response messages

In order to provide consistency across the copies of a context, updates are executed on the primary-copy host. If the primary-copy host of a context is down, a Hybrid Service node advertises the need for selection of primary-copy host of the context with following message: Primary-Copy Selection Request. This message is sent out by multicast by the initiator Hybrid Service node only to those servers holding the permanent-copy of the context under consideration. The purpose of the Primary-Copy Selection Request message is used to select a new primary-copy host if the original is considered to be down. The Primary-Copy Selection Request message is disseminated over a unique topic corresponding to the metadata under consideration. We use the metadata key (UUID) as the topic, which all nodes, holding the permanent-copy of the metadata, within the system subscribe to. By listening to this topic, each existing node receives this message. On receipt of a Primary-Copy Selection Request message, each node response with the Primary-Copy Selection Response message directly to the initiator node. The purpose of this message is to inform the initiator about the permanent-copy of the context under consideration and give some information (such as hostname, transport protocols supported, communication ports) regarding how other nodes should communicate with the answering node. The response message is sent out by unicast directly to the initiator over a unique topic (IP_Address). By listening to this topic, the initiator receives the response message from the answering node.

5.6.4.5 Primary-Copy Notification message

A Hybrid Service node uses a Primary-Copy Notification message to notify the newly selected primary-copy holder. This Notification message is disseminated by unicast directly to the newly selected node. By listening to its unique topic, each existing node may receive a primary-copy notification message, which in turn includes the assignment for being the primary-copy of the context under consideration. Each primary-copy holder of a given context subscribes to a unique topic (such as UUID/PrimaryCopy) to receive messages aimed to the primary-copy holder of that context.

5.6.4.6 Context Update Request and Propagation messages

A Context Update Request message is sent by a replica server to the primary-copy host to ask for handling the updates related with the context under consideration. This message is sent out via unicast by the initiator Hybrid Service node directly to the primary-copy host over a unique topic. By listening to this topic, the primary-copy-host receives the context update request message. A Context Update Propagation message is sent by the primary-copy host only to those servers holding the context under consideration. This message is sent via multicast to the unique topic of the metadata immediately after an update is carried out on the primary-copy to enforce consistency. By listening to this topic, each existing permanent-copy holder node receives a Context Propagation message, which in turn includes the updated version of the context under consideration.

5.6.5 Hybrid Service Discovery Model

The Hybrid Service has a multicast discovery model to locate available services. Initially, a newcomer Hybrid Service sends a multicast Server-Information Request message when it joins the network to make itself available for discovery. Each Hybrid Service network node subscribes to the multicast channel (publicly known topic) to receive Server-Information Request messages. On receiving this request message, each node sends a response message, Server-Information Response message, via unicast directly to the newcomer Hybrid Service. This way, each node makes itself discoverable to other nodes in the system at the bootstrap. Each Hybrid Service node constructs a replica-server-info data structure about other available replica servers in the system. This data structure contains information about decision metrics such as server load and proximity.

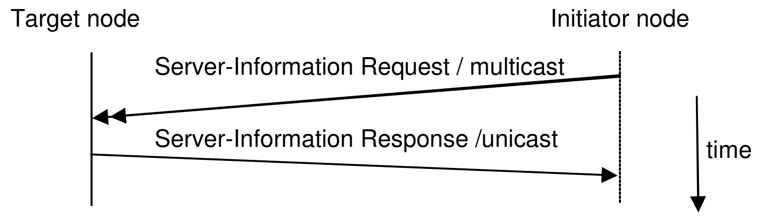

Target node
Initiator node
Server-Information Request / multicast
Server-Information Response /unicast
time

Figure 18 - Message exchanges for Hybrid Service Discovery Model. Each newcomer node sends out a multicast probe message to locate available services in the network. Each target node responds with a unicast message to make themselves discoverable. This figure illustrates the interaction between the initiator server and the target network nodes for service discovery model.

Each node keeps its replica-server-info data structure refreshed. This is done by sending out Server-Information Request messages periodically to obtain up-to-date information. This model enables the system to keep track of proximity and server load information of the available network nodes. This is required for decision-making process of fundamental aspects of the decentralized system architecture such as replica-content placement and consistency enforcement.

5.6.6 Replica Content Placement

In a distributed system, data is replicated to enhance reliability and performance. Replica content placement is a replication methodology that deals with replicating newly inserted data onto other servers, which are capable for storage. After replication, there may only be two types of copies of a context in the system: permanent and server-initiated (temporary). A permanent copy of a context is used as a backup facility to enhance reliability. A server-initiated copy is created temporarily and used to enhance system performance. For the permanent-copy of a context, the Hybrid Service subscribes to a unique topic to receive access/update request concerning the context under consideration. For the server-initiated copy of a context, the Hybrid Service does not subscribe to a topic to minimize the number of messages exchanged for request distribution. The server-initiated copies are only used to enhance the system performance.

In the prototype implementation, the replica content placement process is run offline by the Replication Manager that is responsible of replicating contexts in the system. The Replication Manager runs every so often and checks with the Tuple Pool (in-memory storage) if there are contexts that are newly inserted or updated.

If there is an update, then the update distribution process, which is discussed in Section 5.6.8.1, is performed. After the update distribution process is over, the status of the context is changed from “updated” to “normal”. If there is a newly inserted context in the Tuple Pool, the Replication Manager starts the replica-content placement process (i.e. the distribution of copies of a context into replica hosting environment). This is needed to create certain number (predefined in the configurations file) of permanent replicas. We must note that, on receipt of a client's publish request, an existing node checks if it can handle the request under consideration. Each existing node decides if it is able to store the context by checking the server instantaneous-server-load against the maximum *maximum-server-load-watermark*. Those replica-servers, which are capable of handling the request, perform the operation. However, if the node is overloaded, then this operation is forwarded to the best possible server based on a replica-server selection policy. Figure 19 depicts message exchanges between an initiator Hybrid Service node and a target Hybrid Service node for replica content placement.

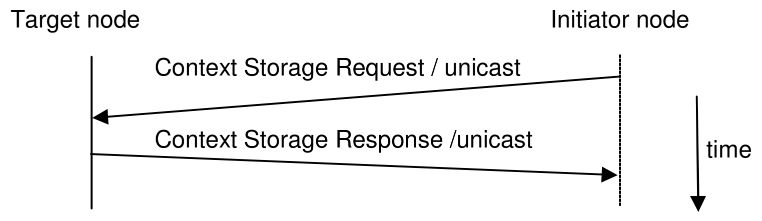

Target node
Initiator node
Context Storage Request / unicast
Context Storage Response /unicast
time

Figure 19 - Message exchanges for Storage (Replica Content Placement). This figure illustrates the interaction between the initiator server and the target network nodes to complete replica-content placement.

Our replica server selection policy takes into account two decision metrics: server load and proximity (see Section 5.6.2). To enforce our selection policy and select replica servers for replica-content placement, we adopt the replica selection algorithm introduced by Rabinovic et al [104] and integrate it with our implementation. The replica server selection process is repeated on target replica servers, until the initiator selects predefined number (*minimum-fault-tolerance-watermark*) of replica servers for replica-content placement. The initiator Hybrid Service chooses the best-ranked server among the selected replica-servers as the primary-copy to enforce consistency.

Once the replica-server selection is completed, the initiator sends unicast message (Context Storage Request message) to the selected replica-servers. On receipt of a storage request, a replica server stores the context as a permanent-copy, followed by sending a response (acknowledgement) message directly to the initiator (via unicast). The newly-selected primary-copy holder receives its Context Storage Request message with a flag indicating that it is the primary-copy holder of that context. Note that, the purpose of storing permanent-copy is for fault-tolerance. The number of permanent replicas is predefined with *minimum-fault-tolerance-watermark* in the configurations file and will remain the same for fault-tolerance reasons. We also utilize the dynamic replication methodology, which is discussed in the next section. This is a performance optimization technique that may move/replicate permanent-copies of a replica onto servers if it is only beneficial for client proximity. This way, the system improves its responsiveness in terms of minimizing the access latency, as the copies of a replica are moved onto servers where the requests are originated.

5.6.7 Dynamic Replication

In order to take into consideration sudden changes in client demands, we use dynamic replication as a performance optimization technique. Dynamic replication deals with the problem of dynamically placing temporary replicas in regions where requests are coming from. This is a push-based replication methodology where a dynamically generated replica is pushed (replicated/migrated) onto a replica server. Such replicas are also referred as push caches [122]. Dynamic replication decisions are made autonomously at each node without any knowledge of other copies of the same data.

In our implementation, we adopt the dynamic replication methodology introduced by Rabinovich et al [104]. This methodology introduces an algorithm, which is used for the Web Hosting Systems, which maintain widely distributed, high-volume, rarely updated and static information. The dynamic replication algorithm by Rabinovich et al considers two issues: a) a replication can take place to reduce the load on a replica server and b) a replication can take place due to changes in the client demands. Our main interest is to provide an optimized performance by replicating temporary-copies of contexts to replica servers in the proximity of demanding clients. To this end, we only focus on the second issue, which concerns with creating replicas if it is only beneficial for client proximity. In the prototype implementation, the dynamic replication process is run by the Dynamic Caching Manager that is responsible for deciding dynamic replica-content placements.

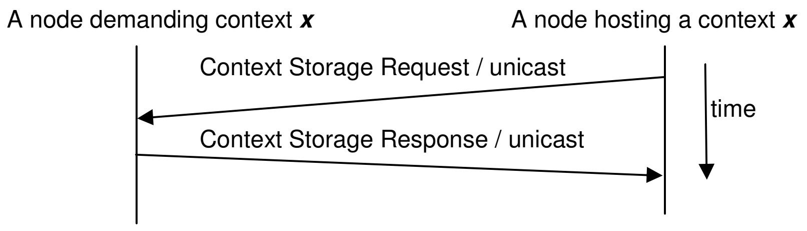

A node demanding context ***x***
A node hosting a context ***x***
Context Storage Request / unicast
Context Storage Response / unicast
time

Figure 20 - Message exchanges for Dynamic Replication/Migration. The dynamic replication/migration process is executed by the Dynamic Caching Manager residing at the initiator node. The Dynamic Caching Manager replicates/migrates data if the demand exceeds certain thresholds. This figure illustrates the interaction between a hosting server and demanding server to complete replica placement/migration for context x.

The Dynamic Caching Manager runs at a Hybrid Service *S* with certain time intervals (*dynamic-replication-time-interval*) and re-evaluates the placement of the contexts that are locally stored. It checks with the Tuple Pool if there are contexts that can be migrated or replicated onto other servers in the proximity of clients that presented high demand for these contexts. It does this by comparing the access request count for each context against some threshold values. If the total demand count for a replica *C* at a Hybrid Service *S* ($cnt_S\ (C)$) is below a ***deletion-threshold(S, C)*** and the replica is a temporary-copy, that replica will be deleted from local storage of Hybrid Service *S*. If, for some Hybrid Service *X*, a single access count registered for a replica *C* at a Hybrid Service *S* ($cnt_S(X, C)$) exceeds a ***migration-ratio***, that service (service *X*) is asked to host the replica *C* instead of service *S*. (Note that the migration-ratio is needed to prevent a context migrate back and forth between the nodes. In our investigation, we chose the *migration-ratio* value as % 60 based on the study introduced in [104]). This means service *S* wants to migrate replica *C* to service *X* which is in the proximity of clients that has issued enough access requests within the predefined time interval (*dynamic-replication-time-interval*). In this case, replica *C* will be migrated to service *X*. To achieve this, a Context Storage Request is sent directly to service *X* by service *S*. On receipt of a Context Storage Request, service *X* creates a permanent copy of the context, followed by sending a Context Storage Response message. If the total demand count for a replica *C* at service *S* ($cnt_S\ (C)$) is above a ***replication-threshold(S, C)***, then the system checks if there is a candidate Hybrid Service, which has requested replica *C*. If, for some Hybrid Service Y, a single access count registered for a replica *C* at service *S* ($cnt_S(Y, C)$) exceeds a ***replication-ratio***, that service (service *Y*) is asked to host a copy of replica *C*. (Note that, in order dynamic replication to ever take place, the replication-ratio is selected below the migration-ratio [104]. In our investigation, we chose the *replication-ratio* value as % 20.) This means service *S* wants to replicate replica *C* to service *Y* that is in the proximity of clients that has issued access requests for this context. An example snapshot of 11-node Hybrid Service replica hosting environment is depicted in Figure 21 where dynamic metadata (contexts ranging from A to O) replicated on the Hybrid Service nodes ranging from 1 to 11. In the example, the quantity of some replicas (for example context replicas D, E and F) is shown more than the quantity of others because of high demand for these replicas. Our aim is not to replicate the server, but the individual contexts based on changing client demands. Figure 20 depicts message exchanges between an initiator node and a target node for dynamic replication process.

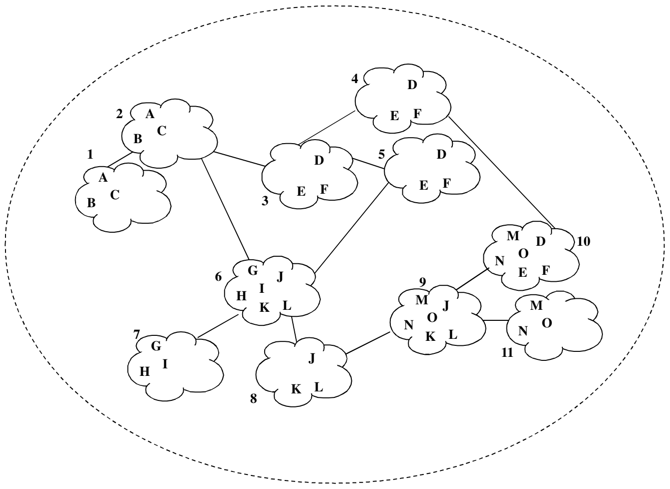


Figure 21 An eleven-node Hybrid Service replica-hosting environment. Numbered callout shapes represent replica servers. Letters ranging from A to O correspond to contexts replicated on the replica servers ranging from 1 to 11. In this example, minimum required degree of replication is two.

5.6.8 Consistency Enforcement

The consistency enforcement issue has to do with ensuring all replicas of a data to be the same. We implement the primary-copy approach for consistency enforcement, i.e., updates are originated from a single site. Tanenbaum classifies this approach as primary-based remote-write protocol [103]. This approach ensures that the primary-copy of a metadata holds up-to-date version of the context under consideration. All update operations are carried out on the primary-copy replica server and the updates are propagated to the permanent-copy holders by the primary-copy.

As mentioned earlier, at any given snapshot of the Hybrid Service network, the system may contain temporary and permanent of copies of a context. On one hand, temporary copies are kept for performance reasons. On the other hand, permanent-copies are kept for fault-tolerance reasons. Each Hybrid Service assigns/creates unique topics for each individual permanent-copy (to receive access and update requests), while it creates no topics for the temporary copies (to avoid flooding the network with access messages). This creates an environment where the system may have different versions of the context, as the temporary copies are not updated. To achieve consistency from the target applications perspective, the Hybrid Service introduces different models to address consistency requirements of different applications. The first model is mainly for read-mostly applications. For these applications, different copies of the context are considered to be consistent and the Hybrid Service allows clients to fetch any copies of the context (permanent or temporary). The second model is for the applications where the update-ratio is high and the consistency enforcement is important. In this case, the Hybrid Service requires the applications to subscribe unique topics of the metadata that they are interested. This way, these applications will be informed of the state changes happening in the metadata immediately after an update occurs. In this model, the primary-copy holder broadcasts the updates through the unique topic corresponding to the metadata under consideration.

We divide the implementation of consistency enforcement into two categories: “update distribution” and “update propagation”. The “update distribution” deals with how the Hybrid Service implements an update operation that take place on the distributed metadata store. The "update propagation" deals with how the Hybrid Service implements the methodology for propagation of updates.

5.6.8.1 Update distribution

On receiving client publication requests, a Hybrid Service node first checks if the request contains a system-defined context key. If not, the system treats the request as if it is a new publication request. In this case, storage process, explained earlier in Section 5.6.6, takes place**.** Otherwise, the system treats publication request as if it is an update request.

The system assigns a synchronized timestamp to each published context (newly written or updated). This is achieved by utilizing NaradaBrokering NTP protocol based timing facility. By utilizing this capability, we give sequence numbers to published data to ensure an order is imposed on the concurrent write operation that take place in the distributed data store. Based on this strategy, a write operation could take place on a data item, only if the timestamp of the updated context was bigger than the version number of the most recent write. This ensures that write/update requests are carried out on a data item x at primary-copy host s, in the order in which these requests are published into the distributed metadata store.

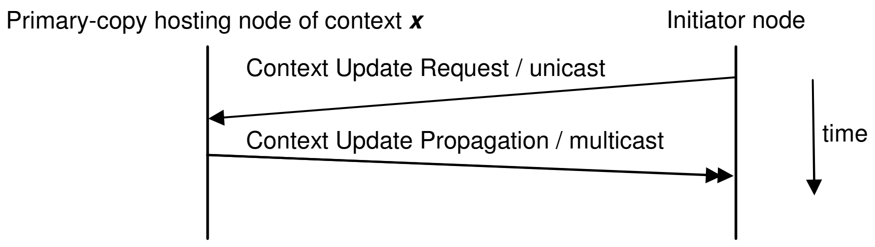

Primary-copy hosting node of context ***x***
Initiator node
Context Update Request / unicast
Context Update Propagation / multicast
time

Figure 22 - Message exchanges for update operation of a context. This figure illustrates the interaction between the initiator server and the primary-copy host node of context x.

An update operation is executed offline, i.e., just after an acknowledgement is sent to the client, by the Replication Manager which is responsible of replicating updated contexts. The update distribution process is executed to perform updates on the primary-copy holder of a context. If the primary-copy host is the initiator node itself, then the update is handled locally. If the primary-copy host is another node, then the update is forwarded to the primary-copy holder. The initiator service sends a message, Context Update Request (see Section 5.6.4.6), by unicast directly to the primary-copy-host for handing over the update handling of a context. The Context Update Request message means that the initiator node is interested in updating the primary-copy replica. This message is sent via unicast by the Replication Manager process offline of the publication request. This message includes the updated version of the context under consideration. On receipt of a Context Update Request message, first, the primary-copy host extracts the updated version of the context from incoming message. Then, it updates the local context if the timestamp of the updated version is bigger than the timestamp of the primary-copy. After the update process is completed, a Context Update Propagation message (see Section 5.6.4.6) is sent to only those servers holding the permanent-copy of the context under investigation. The purpose of the Context Update Propagation is to reflect updates to the redundant copies immediately after the update occurs. On receipt of a Context Update Propagation message from the primary-copy, the initiator Hybrid Service node changes the status of the context under consideration from “updated” to “normal”. If there is no response received from primary-copy host within predefined time interval (*timeout_period*) in response to Context Update Request, the primary-copy host is decided to be down. In this case, the initiator node should select a new primary-copy host (the primary-copy selection process will be discussed in Section 5.6.8.3). After a new primary-copy host is selected, the aforementioned update distribution process is re-executed.

We utilize synchronized timestamps to label published metadata. This allows us to impose an order on the actions that take place in the distributed metadata store. In our implementation, we combine the synchronized timestamps with the primary-based consistency protocol approach. Based on this strategy, each published context is given a synchronized timestamp. An update operation could take place on a data item, only if the timestamp of the newly published update is bigger than the version number of the most recent update. This way, all write operations can be carried out on the primary-copy host, in the same order they were published in to the system. However, this approach has also some practical limits, as the update rate is bounded by the timestamp accuracy of the synchronized timestamps. To achieve ordering among the distributed updates, we use NTP protocol based synchronized timestamps provided by the NaradaBrokering software timing libraries [119].

5.6.8.2 Update propagation

In a distributed data-system, an update propagation process can either be initiated by the server which is in need for the up-to-date copy and wants the pull updates from primary-copy host (pull methodology) or by the server that holds the update and wants to push to other replica servers (push methodology) [109]. In our prototype implementation, we utilized push methodology for update propagation and multicast technique for dissemination of updates. Based on this methodology, whenever an update occurs the primary-copy immediately reflects the changes to the redundant copies in order to keep them up-to-date. Updates can be distributed in two ways: unicast and multicast [103]. In unicast update propagation, the primary copy server sends its updates to replica holders by sending separate messages. In multicast update propagation, it sends its updates using an underlying multicasting facility, which in turn takes care of sending messages to the network. For dissemination of updates, we use the multicast approach and publish the update to the unique topic corresponding to the metadata. This way, the system is able to send the updates only to those permanent-copy holding servers.

5.6.8.3 Primary-copy selection

The primary-copy selection process is used to select a new primary-copy host for consistency enforcement reasons, if the original primary-copy host is down at the moment. A primary-copy host of a context is considered down, if no answer is received in response to a message (such as Context Update Request message) that is directed to it. When the primary-copy host of a context is considered down, the primary-copy selection process is executed step-by-step as depicted in Figure 23 and explained as in the following.

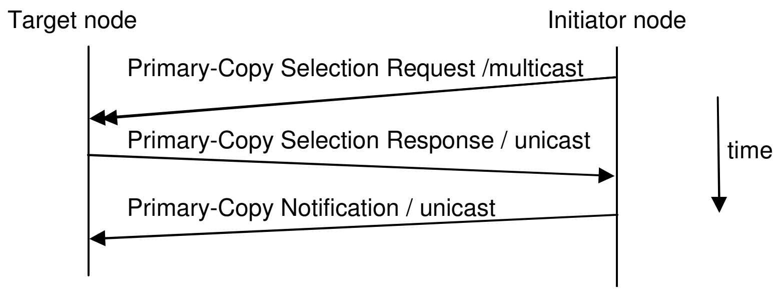

Target node
Initiator node
Primary-Copy Selection Request /multicast
Primary-Copy Selection Response / unicast
Primary-Copy Notification / unicast
time

Figure 23 - Message exchanges for Primary-Copy Selection process. This figure illustrates the interaction between the initiator server and the target network nodes to complete the primary-copy selection process. Time arrow is down.

Say, a Hybrid Service node finds out that a primary-copy of a context is down. In this case, the initiator broadcasts a Primary-Copy Selection Request message (see Section 5.6.4.4) to only those servers holding the context to select the primary-copy host.

On receipt of a Primary-Copy Selection Request message, each replica-holding server that maintains a "permanent" copy of the context under consideration, issues a Primary-Copy Selection Response message (see Section 5.6.4.4). Here, the purpose of a Primary-Copy Selection Response message is to inform the initiator that the answering node contains a permanent copy of the context under investigation. On receipt of the Primary-Copy Selection Response messages, the initiator obtains the information about nodes carrying the permanent copy of the context. Then the initiator selects the best replica server based on a replica server selection process described in Section 5.6.6 as the primary-copy server. In this case, A Primary-Copy Notification message (see Section 5.6.4.5) is sent to the selected server indicating that it is selected as the new primary-copy host for the context under investigation. On receipt of a Primary-Copy Notification message, the permanent-copy holder becomes the primary-copy holder and subscribe the unique address (/UUID/PrimaryCopy) corresponding to the primary-copy of the context under consideration.

5.6.9 Access Request Distribution

On receipt of a client's inquiry request, a Hybrid Service node looks up for the requested context within local storage. If the context exists in local storage, then the inquiry is satisfied and a response message is sent back to the client. If the inquiry asks for external metadata, the system performs the request distribution (access) process, which is discussed in the next section in length.

5.6.9.1 Request Distribution

The prototype implements a request distribution methodology, which is based on broadcast dissemination where the requests are distributed to only those servers holding the context under consideration. This approach does not require keeping track of locations of every single data located in the system. It makes use of copies of a data that are not frequently accessed and kept only for fault-tolerant reasons. In turn, this improves the responsiveness of the system.

Request Distribution: The initiator node issues a Context Access Request message (see Section 5.6.4.2) to the multicast group, if the client's access request is not satisfied in the local storage. This message contains minimum required information (such as context key) regarding the context in demand. The Context Access Request means that the initiator node is interested in discovering the qualified replica servers that may contain the requested context and answer with a response.

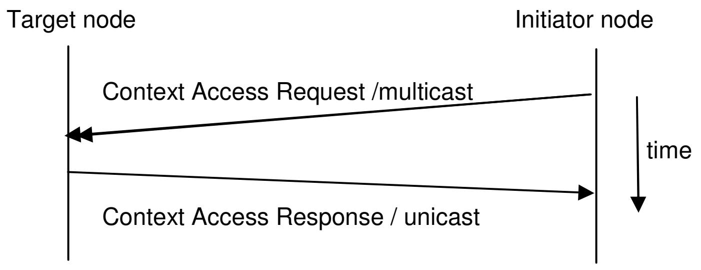

Target node
Initiator node
Context Access Request /multicast
time
Context Access Response / unicast

Figure 24 - Message exchanges for context access. This figure illustrates the interaction between the initiator and a target node hosting the context for request distribution. Time arrow is down.

On receipt of a Context Access Request message, a replica-holding Hybrid Service issues a Context Access Response message (see Section 5.6.4.2). The purpose of a Context Access Response message is to send a response with the context satisfying the query. (Note that, each server keeps track of the count of access requests and the locations where access requests come from for each context. In turn, this enables the system to apply dynamic replication process and adapt to sudden bursts of client demands coming from a remote replica. This is why, if the access request is granted, each server registers the incoming access request in the *access-demanding-server-info* data structure and increments the total *access-request-count* of the context under investigation.) On receiving first Context Access Response message, the initiator Hybrid Service node, obtains the context that can satisfy the query under consideration. Then a response message is sent back to inquiring client. The initiator only waits for responses that arrive within the predefined *timeout* value. If there is no available Hybrid Service node that can satisfy the context query within the *timeout* duration, the access process ends and a “not found” message is sent to the client.

5.7 The WS-Context XML Metadata Service

To support the Hybrid Service for the dynamic, interaction-dependent metadata management requirements of the target application domains, the WS-Context Service prototype was implemented. The WS-Context Service is an implementation of Context Manager component of the WS-Context Specifications. Its main purpose is to provide support for distributed state based systems such as collaboration and workflow-style grids. This service allows the participant’s of an activity to propagate and share context information. With this implementation, we achieved the following capabilities to support the application use domains described in Section 1.3.2.

Firstly, the WS-Context Service implementation introduced a data model and communication protocol for the Context Service component of the WS-Context

Specification. This data model allowed the client applications to store dynamic state metadata based on parent-child relationships that in turn provided flexibility for managing contexts with associations within the WS-Context metadata space. The abstract data models for WS-Context Schema and Query/Publish XML API is discussed in Section 4.2.

Secondly, the WS-Context Service implementation introduced advanced query capabilities to support collaboration grids domain explained in Section 1.3.2.5. Here, the system provided XML API support for enabling real-time playback and session failure recovery capabilities in distributed collaboration session management. The Query/Publish XML API of the WS-Context Service is given in Appendix A.1.

Thirdly, a synchronous callback communication capability is implemented. To utilize this capability a client application has to provide a respondent service, which is used to communicate with the WS-Context Service using synchronous callback style functions. Here, the callback style storage/retrieval functions need to contain the callback address of a respondent service in passing arguments. This allows the WS-Context Service to sent results to a client who initiates a publication or inquiry callback style operations. This functionality is implemented as part of the WS-Context Specifications.

Fourthly, a leasing capability is implemented. This ensured that the out-of-date entries within the WS-Context Service are automatically cleaned up by assigning them an expiration date. This is succeeded by implementing a management scheme for the service entries stored in the database. This scheme implements a leasing manager process, which is responsible for only allowing access to those metadata entries whose leases are not expired and evicting those entries with expired leases from the database.

Fifthly, the WS-Context Service introduced a notification scheme to meet the requirements of collaboration grids. This is succeeded by utilizing a publish-subscribe based messaging scheme. Here, our aim is to inform the interested clients about the state changes happening in a session. On receiving a publishing/deletion/update request for a particular context in a given session, the WS-Context Service multicasts a message. This message includes the type of the operation and the context under consideration and is disseminated over a uniquely identified topic (UUID of the session known by the participants of that session) which all clients participating the session subscribe to. By listening to this multicast group, each client is informed of the state changes happening in that session.

5.8 The Extended UDDI XML Metadata Service

To support the Hybrid Service for the static, interaction-independent metadata management requirements of the target application domains, the extended UDDI prototype was implemented. In order to meet the information requirements of the aforementioned application use scenarios (see Section 1.3.2), the prototype was implemented as a domain independent metadata service. To further support the metadata requirements of Geographical Information Systems [108], this prototype implementation was also extended to support geo-spatial queries on the metadata catalog associated to service entries. With this implementation, we achieved the following capabilities.

Firstly, additional capabilities to existing UDDI Registry are implemented to associate metadata with service entries. Section 4.3 discussed the semantics of both the information model and the programming interface of the extended UDDI service that supports these capabilities. (Also, more detailed information about syntax, arguments, and return values of the extended UDDI programming interface is available in Appendix A.2.).

Secondly, a leasing capability is implemented. This solves a problem with UDDI repositories: information can become outdated, so the out-of-date entries are automatically cleaned up by assigning them an expiration date. This is succeeded by implementing a management scheme for the service entries stored in the database. This scheme implements a leasing manager process, which is responsible for only allowing access to those metadata entries whose leases are not expired and evicting those entries with expired leases from the database. Service providers may extend the lease by updating the metadata entry with the new lease.

Thirdly, Geographical Information System-specific taxonomies are implemented to describe Open Geographical Information System Consortium (OGC) compatible services such as Web Feature Services and their capabilities files. Each Web Feature Service provides data layers corresponding to geographic entities. The "capabilities.xml" file is (in effect) the standard metadata description of OGC services. An important challenge is that existing UDDI Specification does not natively support publishing of services with a bounding box corresponding to a data layer and representing a location of interest. To overcome this problem, we use a standard capability of UDDI registries, tModels, which are used to classify service entries according to predefined taxonomies. For example, we use geographic taxonomies (e.g. the QuadCode taxonomy [72]) to classify UDDI service entries based on spatial coverage. The tModels of the predefined taxonomies are published to the extended UDDI only once at the startup of the system.

This methodology allows us to publish geospatial services based on predefined categories and pose spatial queries on the UDDI-Registry. This way humans and applications can find geospatial services that match a particular data layer with requested spatial coverage. Some examples of these taxonomies used in extended UDDI service can be found at [123].

Fourthly, a dynamic aggregation capability of the geospatial services is implemented to satisfy the metadata requirements of the application usage case discussed in Section 1.3.2.2 in particular. Each Web Feature Service is published into the extended UDDI service based on aforementioned predefined taxonomies. The extended UDDI first checks if the newly published service is actually a geospatial service. Services with the same tModel are services of the same type. If the newly published service entry turns out to include the same tModel as the Web Feature Services do, then the extended UDDI starts interacting with corresponding Web Feature Service to acquire the capabilities file describing the data layers and their spatial coverage. This methodology allows us to provide a Geographical Information System specific metadata catalog registry where the geospatial services can be searched based on their data layers and coverage areas.

Finally, a more general-purpose extension is implemented to the UDDI data model that allows us to insert arbitrary XML metadata into the repository. This may be searched using XPATH queries, a standard way for searching XML documents [124]. This allows us to support other XML-based metadata descriptions developed for other classes of services besides Geographical Information System. The Web Services Resource Framework (WSRF), a Globus/IBM-led effort, is an important example. Our approach allows users to insert both user-defined and arbitrary metadata into the UDDI XML metadata repository.

5.9 Summary

This chapter describes the prototype of Hybrid Grid Information Service. It presented the functional modules and abstraction layers of the system paying particular implementation to design decisions. The implementation of the system is explained under three categories: the Hybrid Grid Information Service, the WS-Context Information Service and the extended UDDI Information Service. As being the focus of the thesis, the Hybrid Grid Information Service implementation was presented within its modular architecture in detail. This architecture consists of Query and Publishing, Expeditor, Filter and Resource Manager, Sequencer, Storage and Access modules. Among them, the implementation of Access and Storage modules was further explained, as these modules implement the fundamental requirements of a replica-hosting environment such as replica-content placement, request distribution, and consistency enforcement.

Chapter 6

Prototype Evaluation

This chapter presents an evaluation of the prototype implementation of the proposed system and investigates its practical usefulness. In this chapter, the following research questions are being addressed:

- What is the baseline performance of the Hybrid Service implementation as far as the WS-Context, extended UDDI and Unified Schema standard operations? (Section 6.2 answers this question.)
- What is the effect of the network latency on the baseline performance of the system? (Section 6.2 answers this question.)
- What is the optimum backup-interval time for achieving high performance and persistency for the standard publication operations? (Section 6.2 answers this question.)

- What is the performance degradation of the system for standard operations under increasing message sizes? (Section 6.3 answers this question.)
- What is the performance degradation of the system for standard operations under increasing message rates? (Section 6.3 answers this question.)
- What is the cost of the access request distribution in terms of the time required to fetch a copy of a data (satisfying an access request) from a remote location? (Section 6.4 answers this question.)
- What is the effect of dynamic replication in the cost of the access request distribution in terms of the time required to fetch a copy of a data? (Section 6.5 answers this question.)
- What is the cost of the storage request distribution for fault-tolerance in terms of the time required to create replicas at remote locations? (Section 6.6 answers this question.)
- What is the cost of consistency enforcement in terms of the time required to carry out updates at the primary-copy holder? (Section 6.7 answers this question.)

6.1 Experimental Setup Environment

We tested our code using various nodes of a cluster located at the Community Grids Laboratory of Indiana University. This cluster consists of eight Linux machines that have been setup for experimental usage. The cluster node configuration is given at Table 5.

Cluster node configuration	
Processor	*Intel® Xeon™ CPU (2.40GHz)*
RAM	*2GB total*
Network Bandwidth	*100 Mbits/sec.*[3] *(among the cluster nodes)*
OS	*GNU/Linux (kernel release 2.4.22)*

Table 5 Summary of the cluster node - machine configurations used in centralized testing experiments

We tested the performance of the prototype with client programs (program for sending queries) and provider programs (program for publishing context). Both clients and providers are multithreaded programs. These applications take following arguments: a) the number of threads and b) number of messages to be fired by each thread. The performance is evaluated with respect to response time at both the querying and publishing 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. 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
```

For the three decentralized setting experiments (such as distribution, fault-tolerance and consistency enforcement), we have selected nodes that are separated by

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

significant network distances. The machines, used in these experiments, are summarized in Table 6.

Summary of Machine Configurations				
	Location	*Processor*	*RAM*	*OS*
gf6.ucs.indiana.edu	*Bloomington, IN, USA*	***Intel® Xeon™ CPU (2.40GHz)***	*2GB total*	*GNU/Linux (kernel release 2.4.22)*
complexity.ucs.indiana.edu	*Indianapolis, IN, USA*	***Sun-Fire-880, sun4u sparc SUNW***	*16GB total*	*SunOS 5.9*
Lonestar.tacc.utexas.edu	*Austing, TX, USA*	***Intel(R) Xeon(TM) CPU 3.20GHz***	*4GB total*	*GNU/Linux (kernel release 2.6.9)*
tg-login.sdsc.teragrid.org	*San Diego, CA, USA*	***GenuineIntel IA-64, Itanium 2, 4 processors***	*8GB total*	*GNU/Linux*
vlab2.scs.fsu.edu	*Tallahase, FL, USA*	***Dual Core AMD Opteron(tm) Processor 270***	*2GB total*	*GNU/Linux (kernel release 2.6.16)*

Table 6 Summary of the machines used in decentralized setting experiments

We used metadata samples (which were actually used in aforementioned Pattern-Informatics application use domain) with a fixed size of 1.7KByte. We illustrate the WS-Context, extended UDDI and Unified Schema metadata samples in Appendix B.1, B.2., and B.3 respectively. Although there is much functionality introduced by the Hybrid Information Service, the focus of the experiments is on key-based publish (save operation) and inquiry (retrieve operation) capabilities.

Analyzing the results gathered from the experiments, we encountered some outliers (abnormal values). Due to outliers, the average may not be representative for the mean value of the observation times. This in turn may affect the results. For example, these outliers may increase the average execution time and the standard deviation. In order to avoid abnormalities in the results, we removed the outliers by utilizing the Z-filtering methodology. In Z-filtering, first, the average and standard deviation values are calculated. Then a simple test is applied. [abs(measurement_i-measurement_average)] / stdev > z_value_cutoff. This test discards the anomalies. After first filtering is over, the new average and standard deviation values are calculated with the remaining observation times. This process was recursively applied until no filtering occurred.

We wrote all our code in Java, using the Java 2 Standard Edition compiler with version 1.5. In the experiments, we used Tomcat Apache Server with version 5.5.8 and Axis software with version 2 as a container. The maximal heap size of the JVM was set to 1024MB by using the option –Xmx1024m. The Tomcat Apache Server uses multiple threads to handle concurrent requests. In the experiments, we increased the default value for maximum number of threads to 1000 to be able to test the system behavior for high number of concurrent clients. As backend storage, we use MySQL database with version 4.1. We used the "nanoTime()" timing function that comes with Java 1.5 software.

6.2 Responsiveness Experiment

The primary interest in doing this experiment is to understand the baseline performance of the implementation of the Hybrid Service. The performance evaluation of the service is done for publish functions under normal conditions, i.e., when there is no additional traffic. For the responsiveness experiment, the aim was to explore the optimal performance of the system on a centralized setting. Here, the Hybrid Service was running on cluster node-6, while the client and provider applications were running on the cluster node-5. One should keep in mind that given client/server architecture, with all machines on the same network, is setup to measure an approximation of the optimal system performance. We expect that the results measured in this environment will be the optimal upper bound of the system performance.

In this experiment, we particularly investigate performance of our in-memory storage methodology for the extended UDDI, WS-Context and Unified Schema standard operations. We conduct following testing cases: a) A client sends publish requests to an echo service. The echo service receives a message and then sends it back to the client with no processing applied. b) A single client sends publish requests to a Hybrid Service where the system grants the request with memory access. c) A single client sends publish requests to a Hybrid Service where the system grants the request with database access.

This experiment studies the effect of various overheads that might affect the system performance. To do this, an echo service is used. The echo service returns the input parameter passed to it with no processing applied. This service helps measuring various overheads such as the network communication, client application initialization and container processing. By comparing and contrasting the results from the echo service and the Hybrid Service, the actual time spent for pure server side processing can be observed. In this experiment, we use the same Web Service container engine (Apache Axis with version 2) for all testing cases.

In our investigation of system performance, we conducted the testing cases when there were 5000 metadata published in the system. At each testing case, the client sends 200 sequential requests for publish purposes. We record the average response time. This experiment was repeated five times. The design of these experiments is depicted in Figure 25.

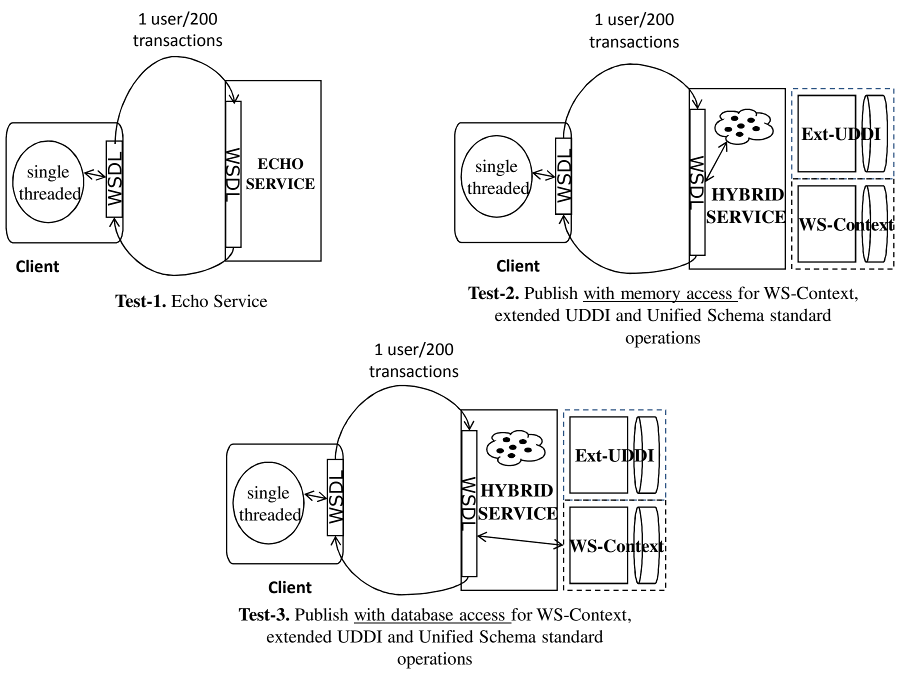


Figure 25 Testing cases of responsiveness experiment for a standard operation

6.2.1 Results of the Responsiveness Experiment

We conduct an experiment where we investigate the best possible backup-interval period to provide persistency and high performance at the same time. Here, for testing purposes, we used WS-Context Schema primary operations: save_context and get_context. Based on this experiment, we observe the trade-off in choosing the value for backup-time-interval. If the backup frequency is too high such as every 10 milliseconds, then the time required for a 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 our experiments.

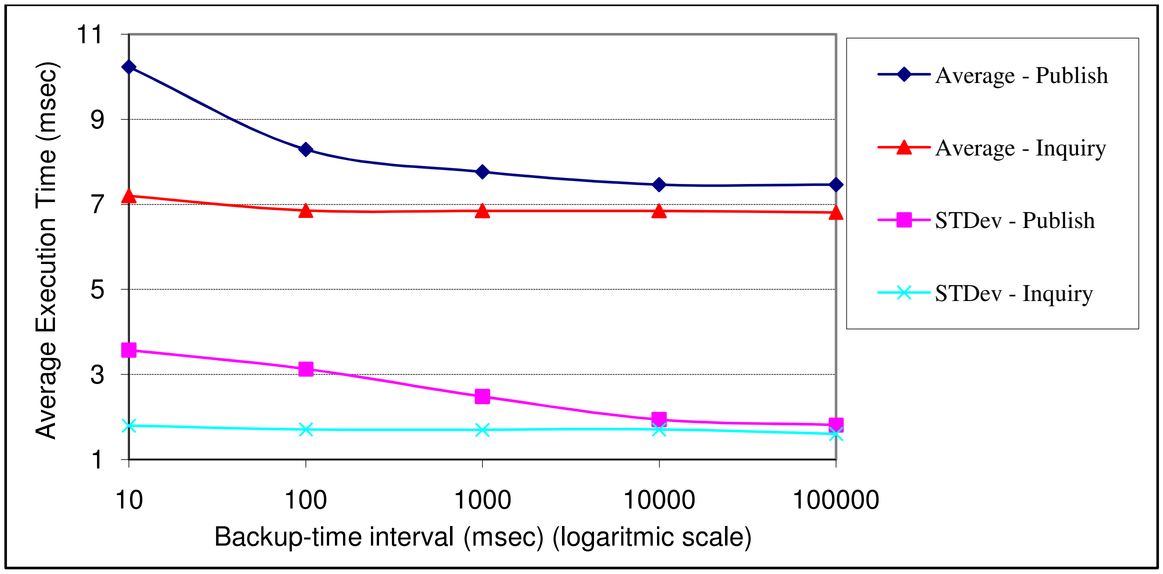


Figure 26 Test results for backup frequency investigation

	Publish Function		Inquiry Function	
Interval Time (seconds)	Average timings (msec)	STDev (msec)	Average timings (msec)	STDev (msec)
0.01	10.24	3.57	7.20	1.80
0.1	8.29	3.13	6.86	1.71
1	7.76	2.48	6.85	1.70
10	7.46	1.94	6.85	1.71
100	7.46	1.82	6.81	1.60

Table 7 Statistics for the Figure 26

Figure 27 and Figure 28 show the performance results of publish operation of Unified Schema and WS-Context Schema. We publish WS-Context type metadata, which is interaction-dependent metadata, with either WS-Context or Unified Schema publish operations. Similarly, Figure 29 and Figure 30 show the performance results of publish operation of Unified Schema and extended UDDI Schema. We publish UDDI-type metadata, which is interaction-independent metadata, with either extended UDDI or Unified Schema publish operations.

The results show that we gain more than 50% performance increase by employing an in-memory storage mechanism in our design. This experimental study indicates that one can achieve noticeable performance improvements in metadata management for standard operations by simply employing an in-memory storage mechanism, while preserving a certain persistency level, as the metadata have to be backed up offline in at most N time unit. Based on our investigation on backup frequency, we choose the value of N to be every 10 seconds.

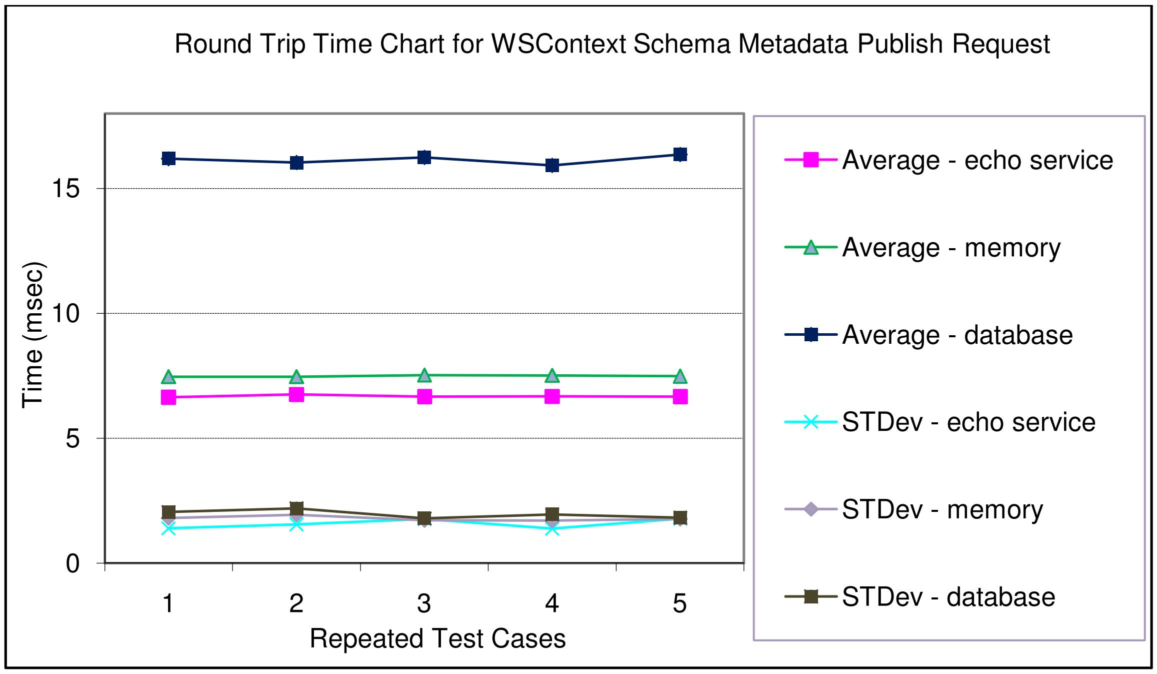


Figure 27 Round Trip Time Chart for WSContext Schema Metadata Publish Requests

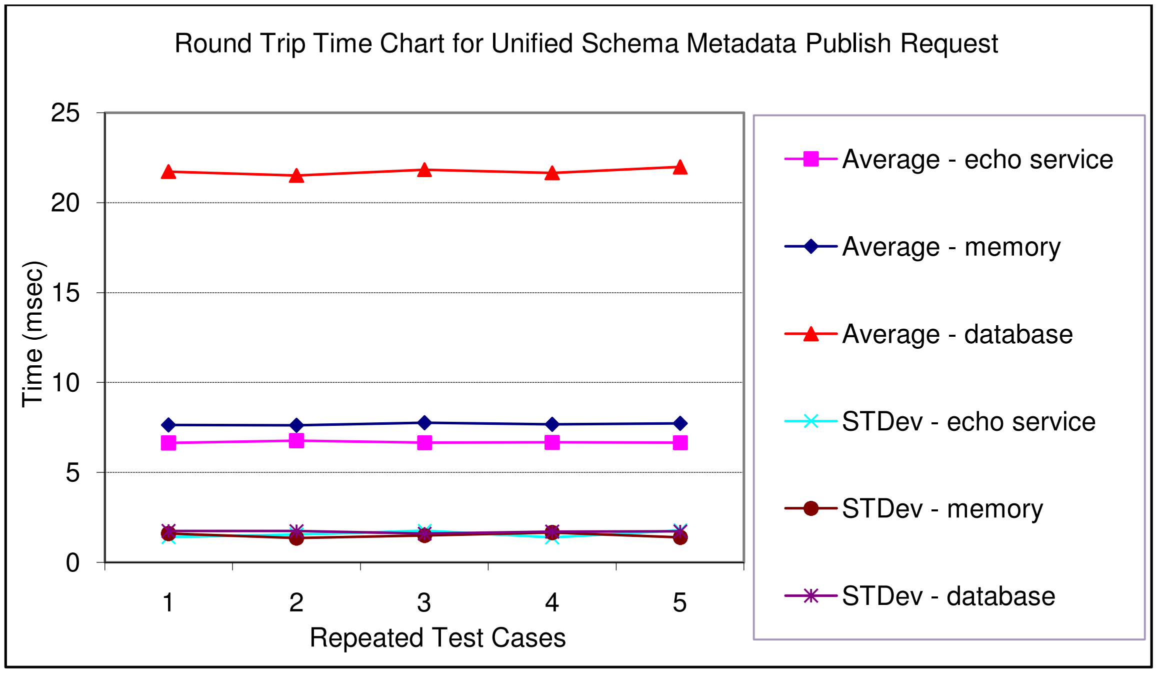


Figure 28 Round Trip Time Chart for Unified Schema Metadata Publish Requests for publishing WS-Context type metadata

Statistics for the first test set from different publish request testing cases		
	Average timings	STDev
Test-1 – Echo Service	6.64	1.40
Test-2 – Unified - memory	7.46	1.82
Test-3 – WSContext - memory	7.77	1.51
Test-4 – Unified - database	21.73	1.75
Test-5 – WSContext - database	16.24	1.80

Table 8 Statistics for the first test set. We conduct testing cases to learn performance of the Unified and WS-Context Schema standard publish operations. In these tests, we publish WSContext-type (interaction-dependent) metadata with Unified Schema publish operation and WSContext Schema publish operation through the Hybrid Service. (Test-1: Echo service testing case, Test-2: Unified Schema publish-operation with memory access testing case, Test-3: WS-Context 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.

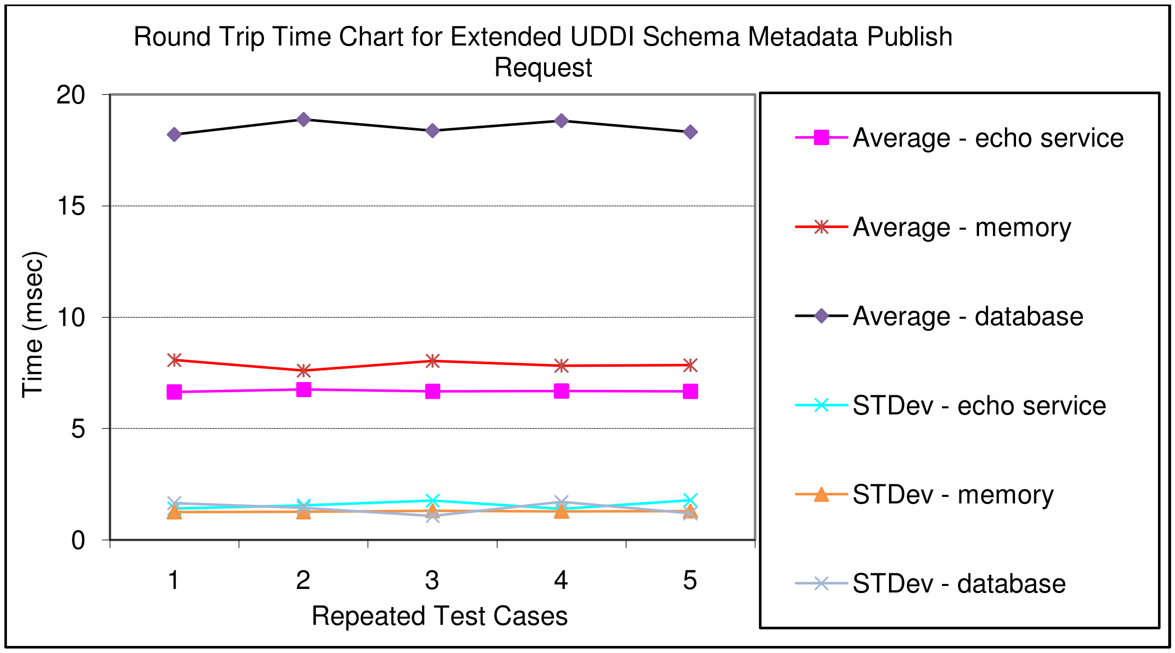


Figure 29 Round Trip Time Chart for Extended UDDI Metadata Publish Requests

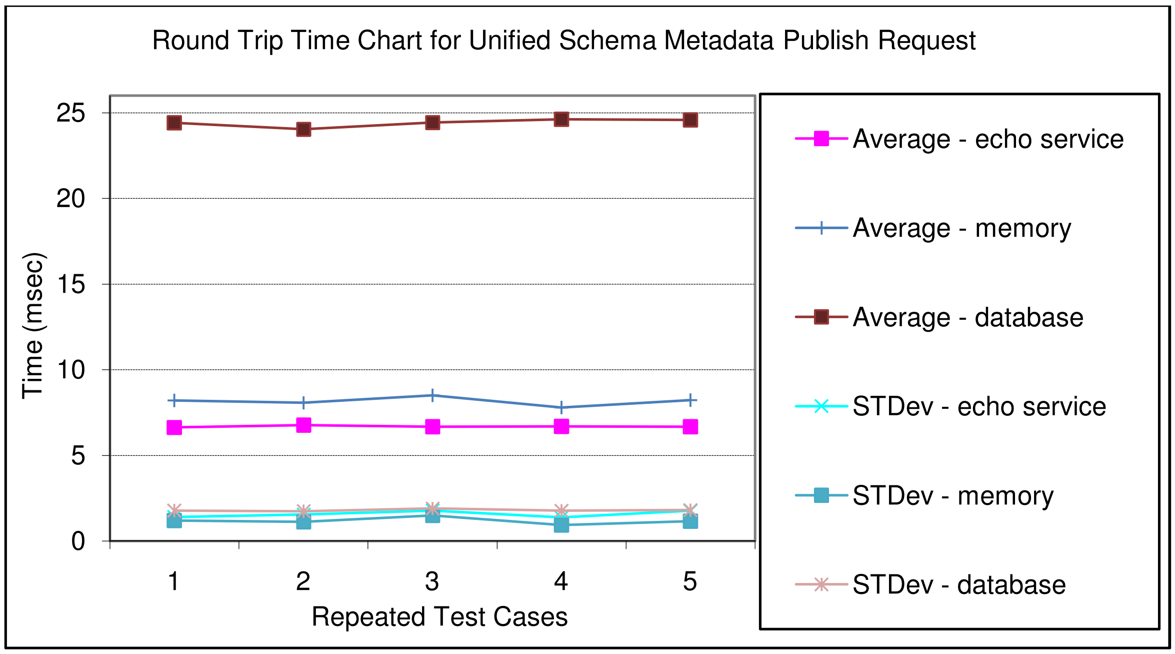


Figure 30 Round Trip Time Chart for Unified Metadata Publish Requests for publishing UDDI-type metadata

Statistics for the second test set from different publish request testing cases		
	Averate timings	STDev
Test-1 – Echo Service	6.64	1.40
Test-2 – Unified - memory	8.08	1.13
Test-3 – Ext UDDI - memory	7.61	1.26
Test-4 – Unified - database	24.41	1.78
Test-5 – Ext UDDI - database	18.88	1.44

Table 9 Statistics for the second test set. We conduct testing cases to learn performance of the Unified Schema and extended UDDI Schema standard publish operations. In these tests, we publish UDDI-type (interaction-independent) metadata with Unified Schema publish operation and extended UDDI Schema publish operation through the Hybrid Service. (Test-1: Echo service testing case, Test-2: Unified Schema publish-operation with memory access testing case, Test-3: Extended UDDI Schema publish-operation with memory access testing case, Test-4: Unified Schema publish-operation with database access testing case, Test-5: Extended UDDI Schema publish-operation with database access testing case). The time units are in milliseconds.

6.3 Scalability Experiment

In this experiment, we conducted two testing cases on the centralized version of the Hybrid Service to investigate its scalability. We tried to answer the following two questions: a) how well does the system perform when the context size is increased, b) how well does the system perform when the message rate per second is increased.

In the first testing case, our goal is to quantify the performance degradation in response time when contexts, with larger sizes, published/retrieved into/from the Hybrid Service. We have done this by increasing the context sizes until the response time degrades. In this experiment, round-trip time was recorded at each inquiry/publish request message. To facilitate the testing, we used WS-Context Schema publish and inquiry operations on the Hybrid Service. The design of this testing case (Test-5) is depicted in Figure 31.

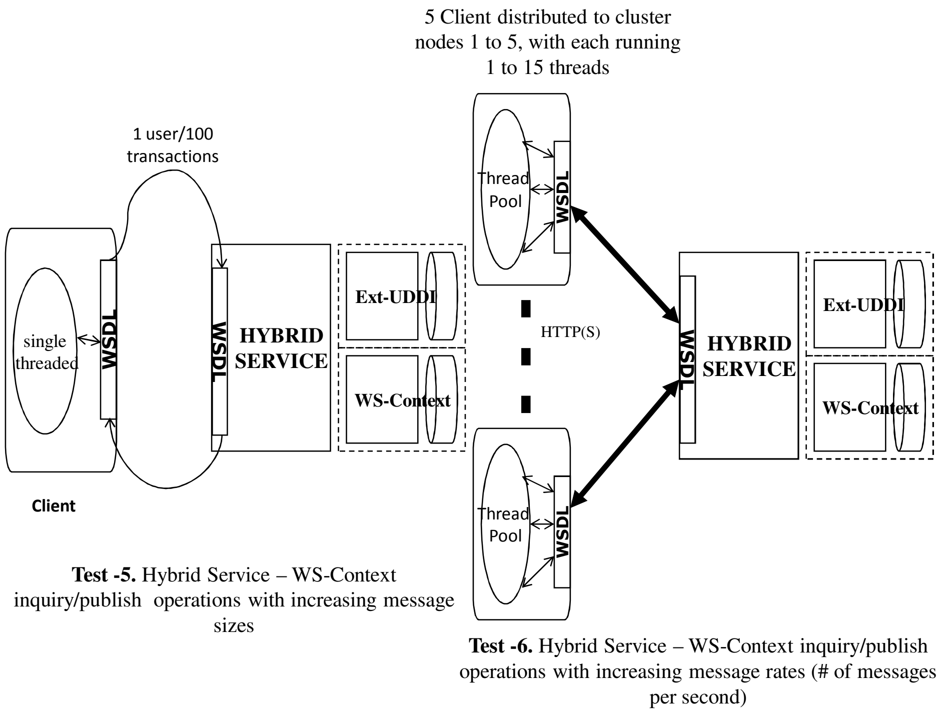

5 Client distributed to cluster nodes 1 to 5, with each running 1 to 15 threads
1 user/100 transactions
single threaded
WSDL
WSDL
HYBRID SERVICE
Ext-UDDI
WS-Context
Client
Thread Pool
WSDL
HTTP(S)
HYBRID SERVICE
Ext-UDDI
WS-Context
Thread Pool
WSDL
Test -5. Hybrid Service – WS-Context inquiry/publish operations with increasing message sizes
Test -6. Hybrid Service – WS-Context inquiry/publish operations with increasing message rates (# of messages per second)

Figure 31 Testing cases of scalability experiment for inquiry and publish functionalities

In the second testing case (Test-6), we want to determine how well the number of users anticipated can be supported by the system for constant loads. Our goal is to quantify the degradation in response time at various levels of simultaneous users. In order to understand such performance degradation, we evaluate standard Hybrid Grid Information Service operations. To facilitate the testing, we use WS-Context standard operations with additional concurrent traffic. We have done this by ramping-up the number of messages sent per second until the system performance degrades. In this experiment, messages are fired off in random fashion. In order to ensure randomness of message distribution, the client applications are scattered into five different machines. To synchronize all clients (located in different machines) to start/stop firing messages at the same time, publish-subscribe based methodology is used. By listening to a predefined topic, each client receives "start/stop firing" message, which in turn starts/stop the testing process and synchronize the clients distributed into different machines. To increase the message rate, both number of iterations and number of threads at each client (in each machine) are gradually increased. In order to minimize the influence of thread scheduling on the latency, the number of threads, at each machine, range from 1 to 15. In this experiment, the Hybrid Service was running on cluster node-6, while the client and provider were running on the cluster nodes ranging from node-1 to node-5. We recorded the round trip time at each inquiry/publish request message and applied this test for both publish and inquiry standard operations. The design of this test is depicted in Figure 31, while the results are depicted in Figure 34. The detailed statistics are given in Table 12.

6.3.1 Results of the Scalability Experiment

Based on the results, we note that Hybrid Grid Information Service standard operations performed well for increasing context sizes. For example, Figure 32 indicates that the cost of inquiry and publish operations remains almost the same, as the context's payload size increases from 100Bytes up to 10KBytes. Figure 33 indicates the system behavior for publish message for the context payload size between 10Kbytes and 100Kbytes. By comparing the results from an Echo Service and Hybrid Service, we observe that the pure server processing time remains the same as the size of the messages increase.

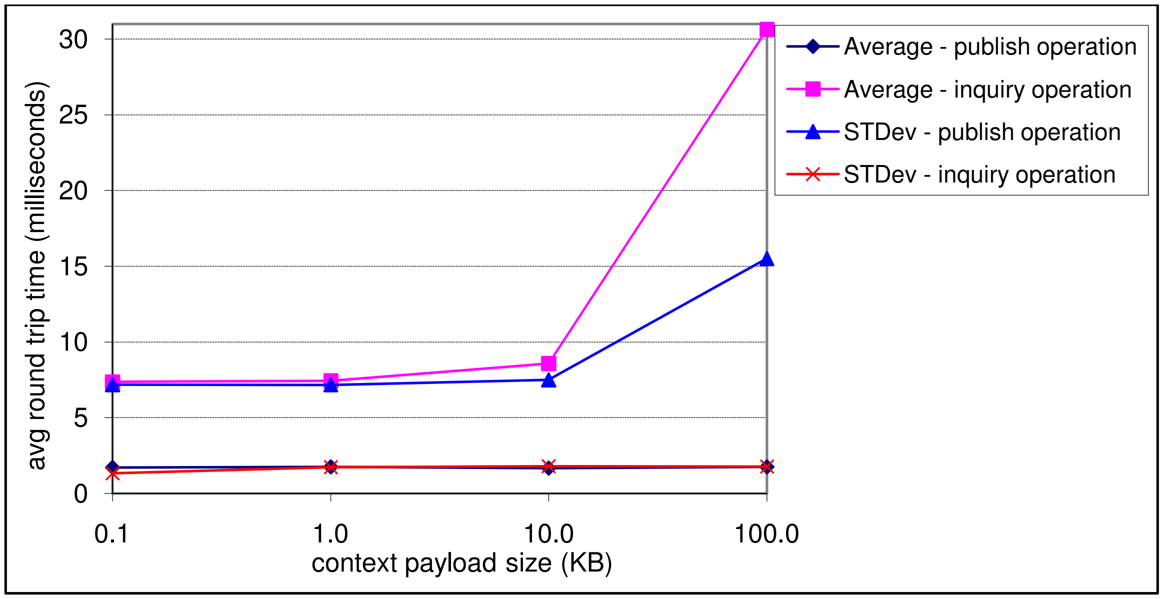


Figure 32 Logarithmic scale round trip time chart for Hybrid Service - WS-Context inquiry and publish operations when context payload size increases

	inquiry operation - memory access		publish operation - memory access	
Kbytes	Average timings	STDev	Average timings	STDev
0.1	7.18	1.34	7.38	1.70
1	7.17	1.73	7.43	1.75
10	7.50	1.79	8.58	1.67
100	15.50	1.77	30.64	1.93

Table 10 Statistics of Figure 32 for Hybrid Service - WS-Context Schema API - inquiry and publish operations with changing context payload sizes. Time units are in milliseconds.

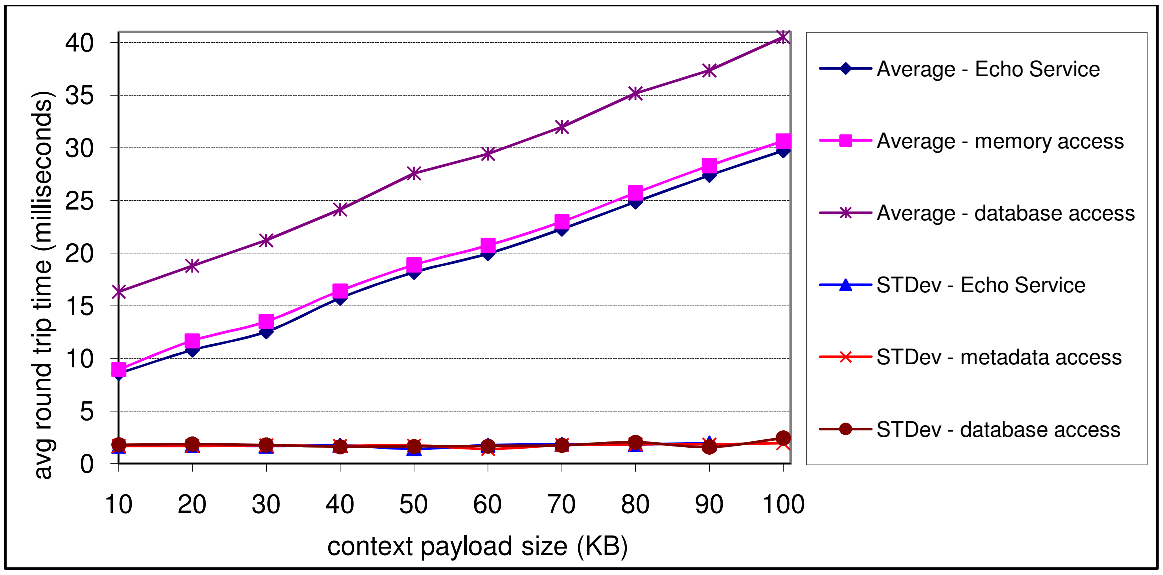


Figure 33 Round Trip Time chart for publish requests when context payload size increases from 10Kbytes to 100Kbytes

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

Table 11 Statistics of Figure 33 for Hybrid Service - WS-Context Schema - publish operations with changing context payload sizes. Time units are in milliseconds

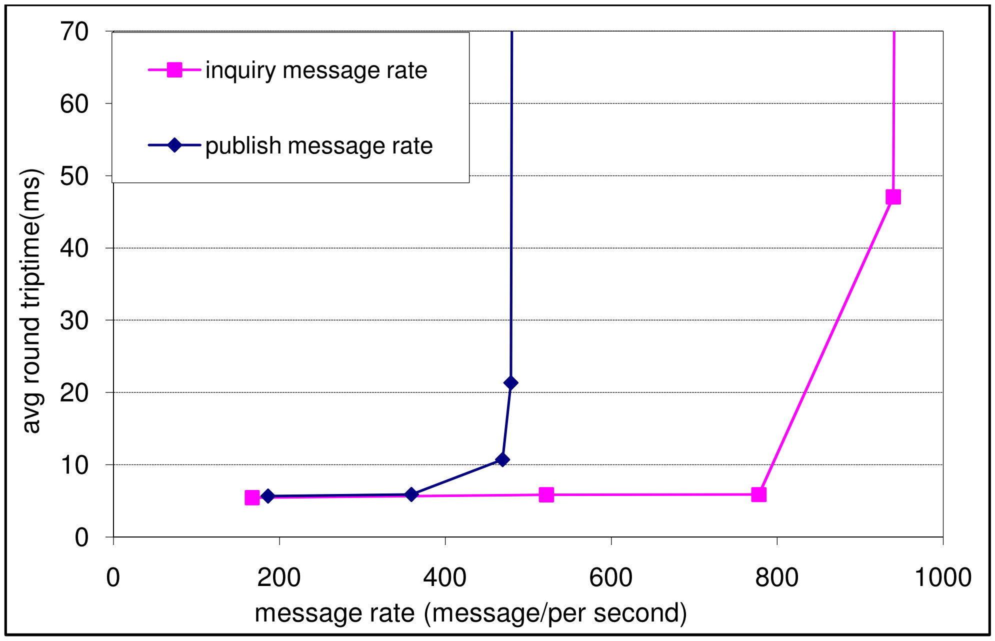


Figure 34 Average Hybrid Service – WSContext Schema inquiry and publish response time chart - response time at various levels of message rates per second

Hybrid Service - WS-Context Schema inquiry operation		
messages/second	Average timings	STDev
167	5.45	0.65
522	5.84	0.97
778	5.9	0.91
940	47.05	33.52
942	92.25	45.13
Hybrid Service - WS-Context Schema publish operation		
messages/second	Average timings	STDev
186	5.65	2.07
359	5.86	2.94
469	10.69	8.28
479	21.36	16.51
480	70.57	52.22

Table 12 Statistics of the experiment results depicted in Figure 34. These measurements were taken with Hybrid Service when the WS-Context Schema inquiry and publish request is granted with memory access. Time units are in milliseconds.

Based on the results depicted in Figure 34 and listed in Table 12, we determine that a large number of concurrent inquiry requests may well be responded to without any error by the system and do not cause significant overhead on the system performance. We observe that after around 800 inquiry messages per second, the system performance degradates 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. Based on the results depicted in Figure 34 and listed in Table 12, we also determine that a significant number of concurrent publication requests may well be responded without any error by the system and do not cause big overhead on the system performance. We observed that the system performance starts dropping down after around 400 publication messages per second within a second. This threshold is mainly due to the persistency capability of the system. As the publish message-rate is increased, the number of updated/newly written contexts (within a unit time interval) in the Tuple Pool is also increased. In turn, the time required for writing the larger number of updates into local information service back-end is increased. Thus, we see higher fluctuations in the response times for increasing number of simultaneous publish requests by examining the standard deviations results listed in Table 12.

6.4 Distribution Experiment

In this experiment, we conducted various testing cases to investigate the cost of distribution. We measured the cost of distributing access request into remote servers separated with significant network distances.

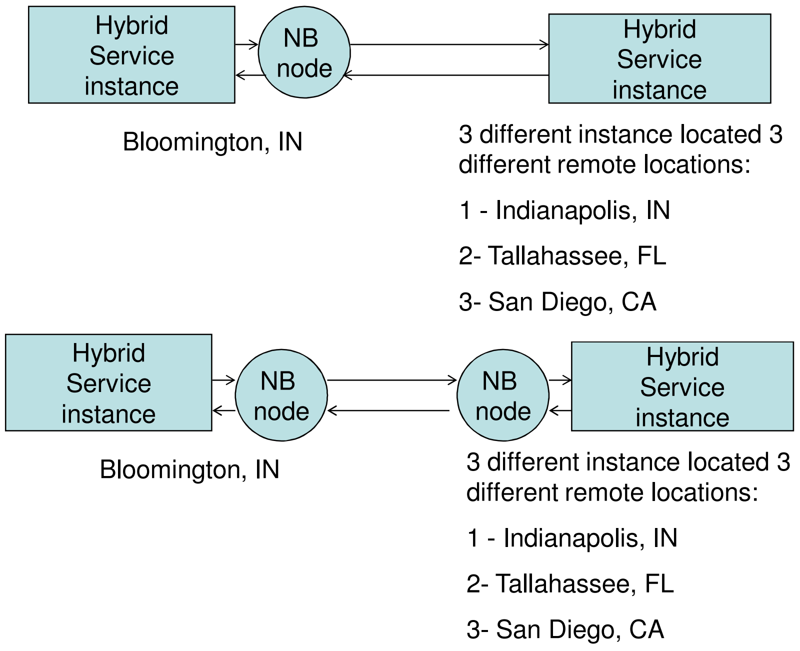


Figure 35 The design of the distribution experiment. The rounded shapes indicate NaradaBrokering nodes. The rectangle shapes indicate Hybrid Service instances located at different locations. The first test was conducted with one broker where the broker is located before the Hybrid Service instance in Bloomington, IN, while the second test was conducted with two broker nodes each sitting on the same machine before the Hybrid Service instance.

In particular, we performed this experiment to answer following questions: a) what is the cost of access request distribution in terms of time required to fetch copies of a data (satisfying an access query) from remote locations?, b) how does the cost of distribution change when using multiple intermediary brokers for communication?, c) how does the performance of the distribution change for continuous, uninterrupted operations?

6.4.1 Results of the Distribution Experiment

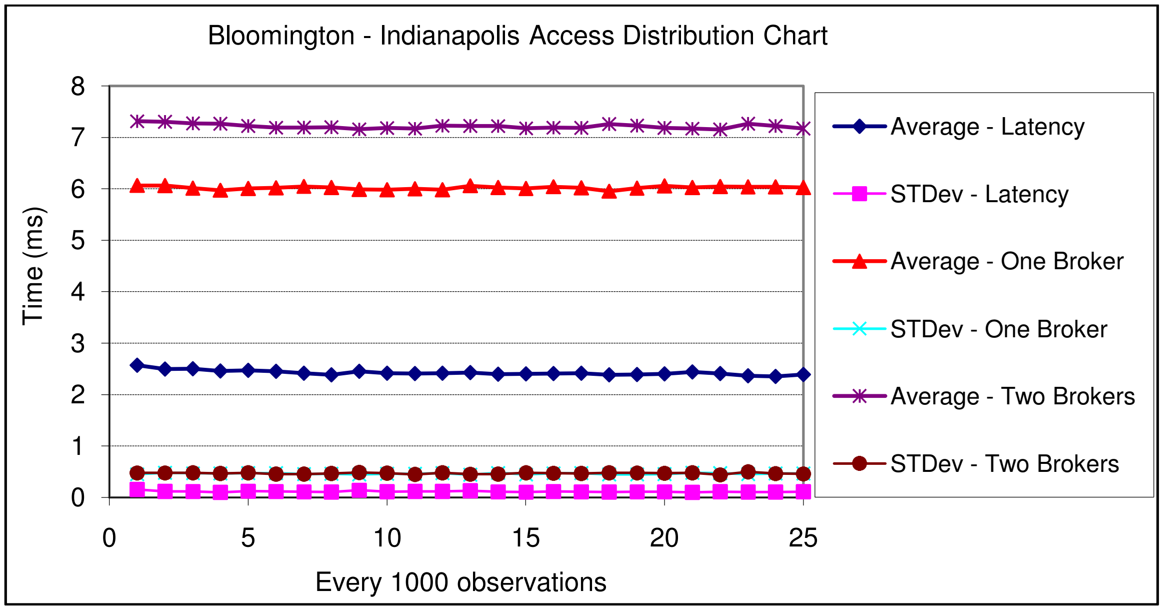


Figure 36 The Distribution Experiment Results between Bloomington and Indianapolis - Each point in the graph corresponds to average of 1000 observations.

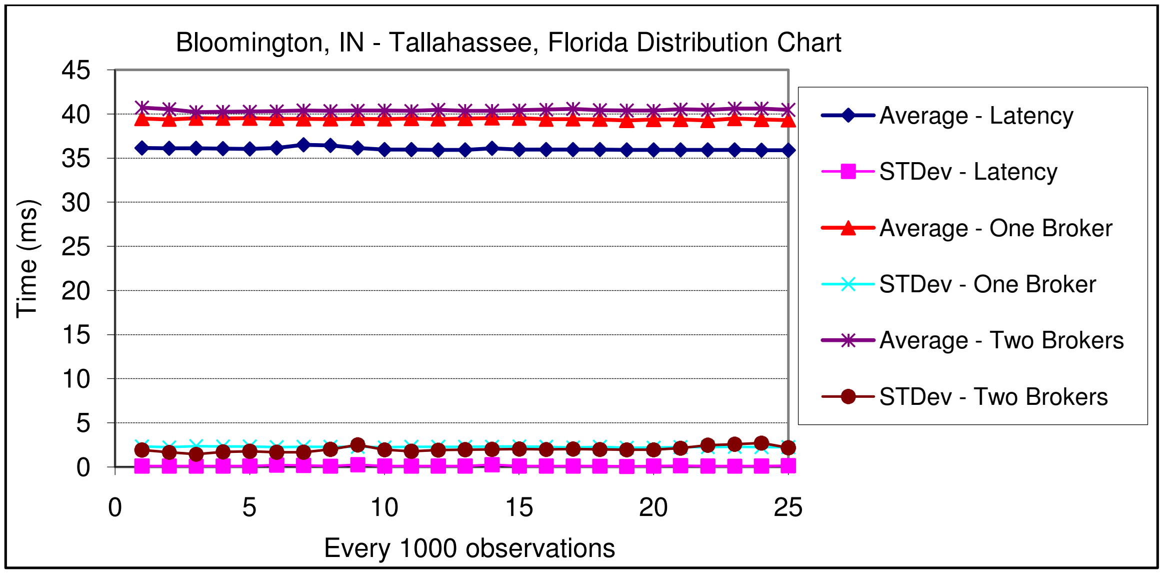


Figure 37 The Distribution Experiment Results between Bloomington and Tallahasse - Each point in the graph corresponds to average of 1000 observations.

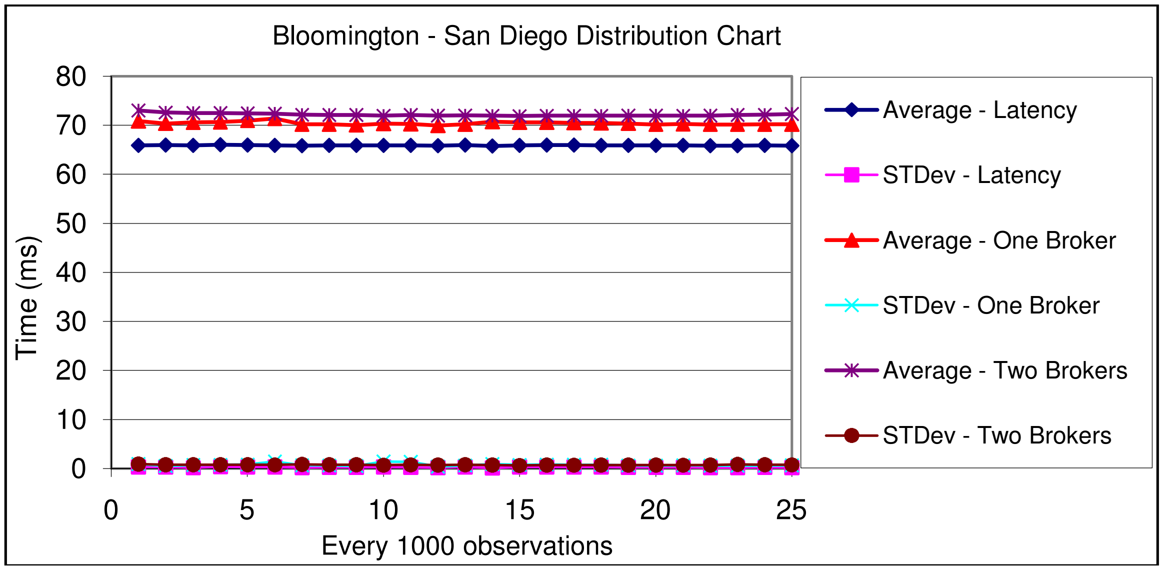


Figure 38 The Distribution Experiment Results between Bloomington and San Diego - Each point in the graph corresponds to average of 1000 observations.

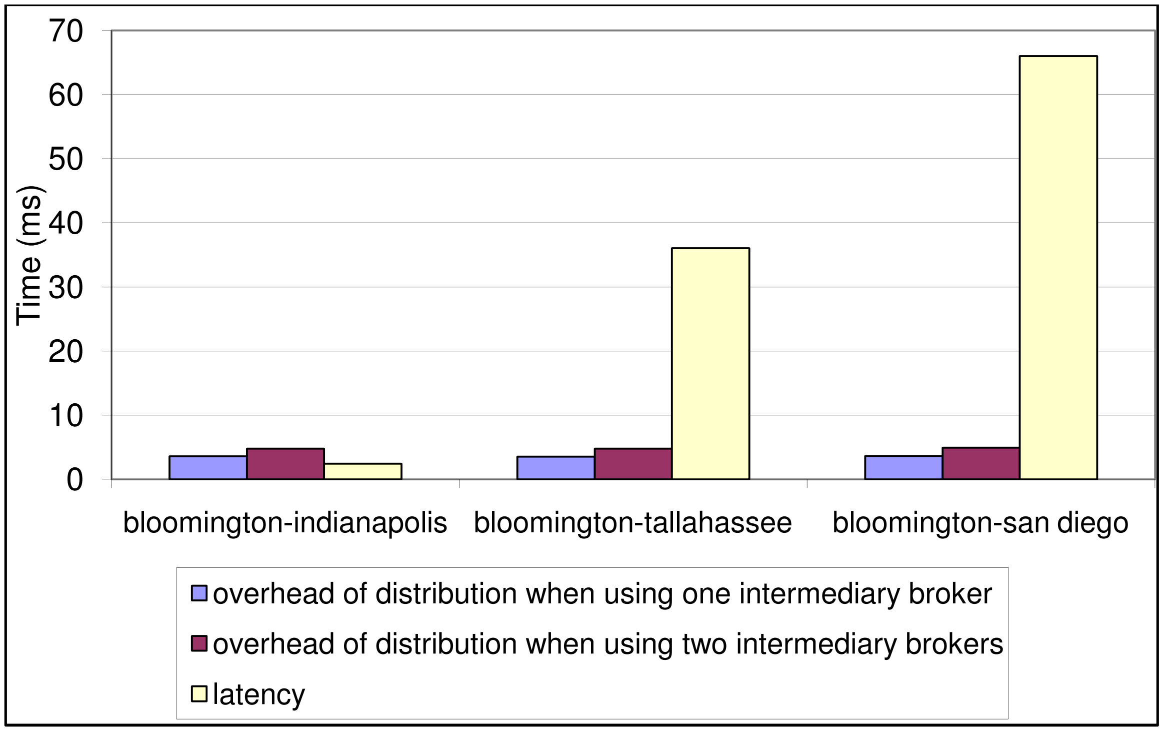


Figure 39 Time spent in various sub-activities of the request distribution scheme of the Hybrid Service

	one broker	two brokers	latency
bloomington-indianapolis	3.59	4.79	2.42
bloomington-tallahassee	3.55	4.78	36.05
bloomington-san diego	3.63	4.92	66

Table 13 Statistics for Figure 39. Overhead of request distribution. Average timings in milliseconds.

Based on the results depicted in Figure 36, Figure 37, and Figure 38, we extract the processing time involved for access request distribution. We depict the time spent in various sub-activities of distribution in Figure 39 and list the results in Table 13. By analyzing the results, we observe that regardless of how the Hybrid Service instances are distributed, the system showed the same stable performance, which is around 3.6 ms when using one intermediary broker. This time includes the Hybrid Service system processing overhead and overhead of using an intermediary broker as part of publish-subscribe system. We observe that the overhead of access request distribution increases only by 1.2 ms when we use an additional intermediary broker. The results also indicated that the system performs well for continuous, uninterrupted request distribution operations.

6.5 Dynamic Replication Experiment

In this experiment, we conducted a testing case to investigate the performance of dynamic replication. We used the dynamic replication for performance optimization to replicate temporary copies of contexts to where they wanted. In this experiment, we simulated a workload, where we have 1000 metadata in the Hybrid Service instance located at Indianapolis, IN. The size of the metadata is around 1.7 KByte. The dynamic replication placement decision takes place every 100 seconds. The dynamic replication deletion threshold was 0.03 requests per second, while the replication threshold was 0.18 requests per second. In this testing case, metadata from the Indianapolis instance was requested randomly by the Hybrid Service instance located at Bloomington. If the remote metadata is replicated to local site, the system simply obtains the data from local in-memory storage. We conducted two testing cases to answer the following questions: a) What is the cost of access distribution to fetch copies of a context from the remote location (Indianapolis), when the dynamic replication is disabled, b) What is the cost of access distribution to fetch copies of a context from the remote location (Indianapolis), when dynamic replication is enabled.

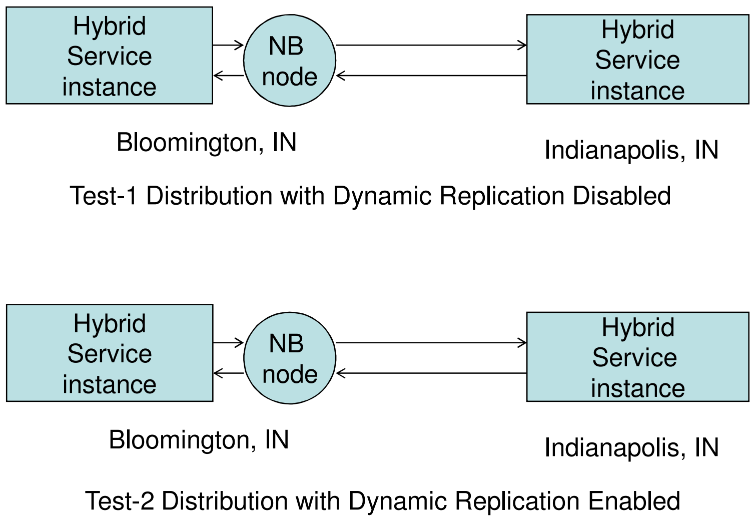


Figure 40 The design of the dynamic replication experiment. The rounded shapes indicate NaradaBrokering nodes. The rectangle shapes indicate Hybrid Service instances located at different locations. In the first testing case, dynamic replication capability is disabled. In the second testing case, dynamic replication capability is enabled.

6.5.1 Results of the Dynamic Replication Experiment

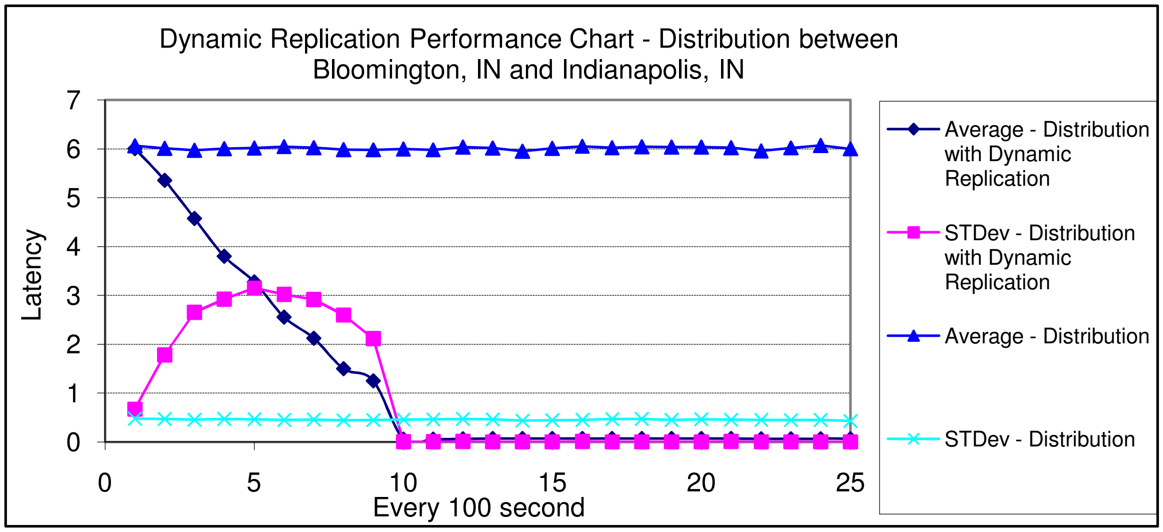


Figure 41 The results of the dynamic replication experiment.

Based on the results depicted in Figure 41, in this experiment, we observed that the dynamic replication methodology could actually move highly requested metadata to where they wanted. We observed that the system stabilized after around 16 minutes. Here, the system managed to move half of the metadata to the local site after around 8 minutes, where we observed the highest peak in the standard deviation values. This is simply because half of the access requests were granted locally, while the other half were granted at the remote location.

6.6 Fault-tolerance Experiment

In this experiment, we conducted various testing cases to investigate the cost of fault-tolerance when moving from centralized system to a decentralized replica hosting system. In particular, we performed our testing cases to answer following questions: a) What is the cost of replica-content placement for fault-tolerance in terms of the time required to create replicas at remote locations?, b) How does the system behavior change for continuous, uninterrupted replica-content placement operations?. To answer these questions, we conducted two testing cases: The first test was conducted with one broker when the broker was located before the Hybrid Service instance at Bloomington, IN. The second test was conducted with two brokers each sitting on the same machine before the Hybrid Service instances. In this experiment, we increased the fault tolerance level gradually and measured end-to-end latency for replica-content placement.

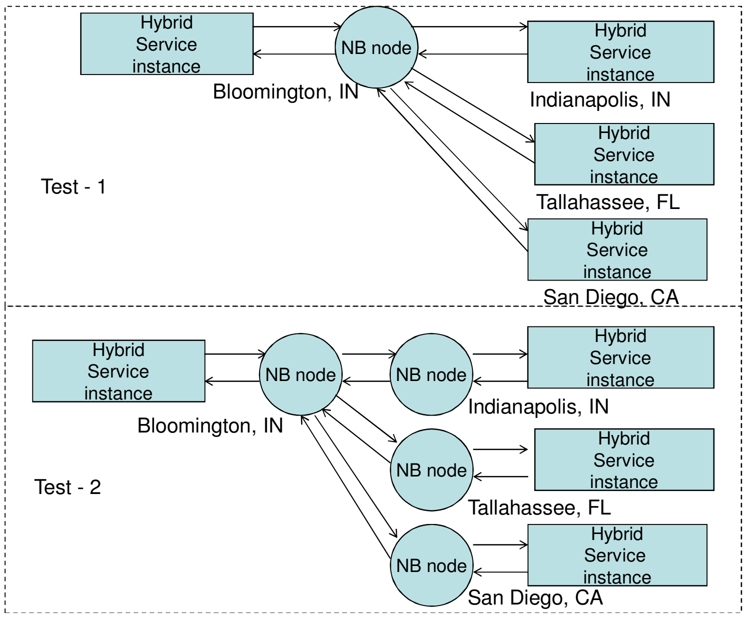


Figure 42 The design of the fault tolerance experiment. The rounded shapes indicate NaradaBrokering nodes. The rectangle shapes indicate Hybrid Service instances located at different locations. In the first testing case, we measure the end-to-end latency for varying number replica-content creation with only one broker. In the second case, we repeat the same test with two brokers.

6.6.1 Results of the Fault-tolerance Experiment

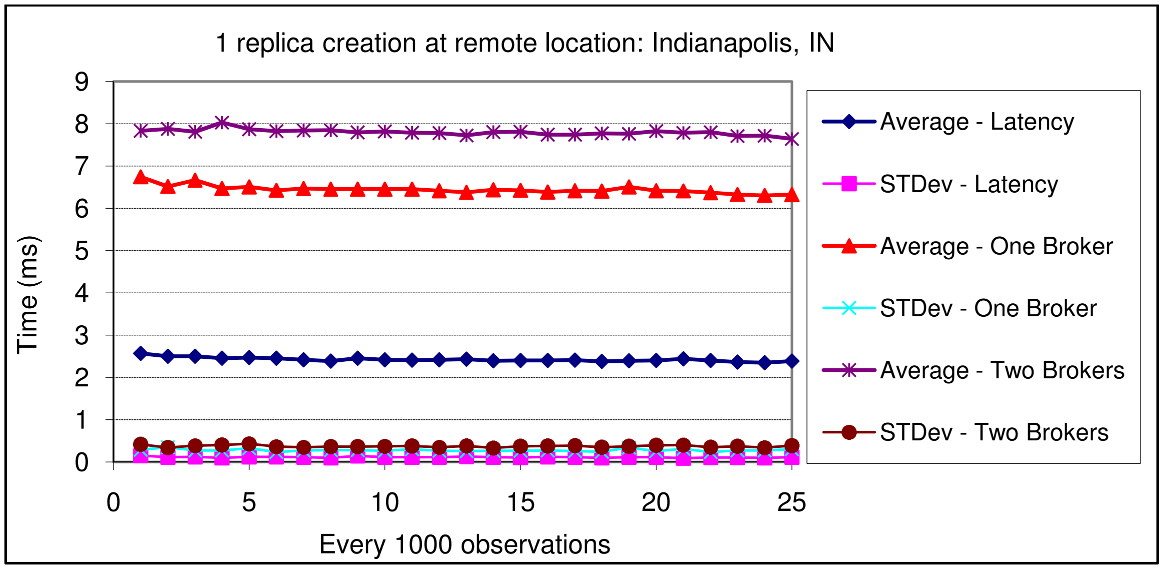


Figure 43 Fault Tolerance Experiment results when one replica is created at Indianapolis, IN. Each point in the graph corresponds to average of 1000 observations.

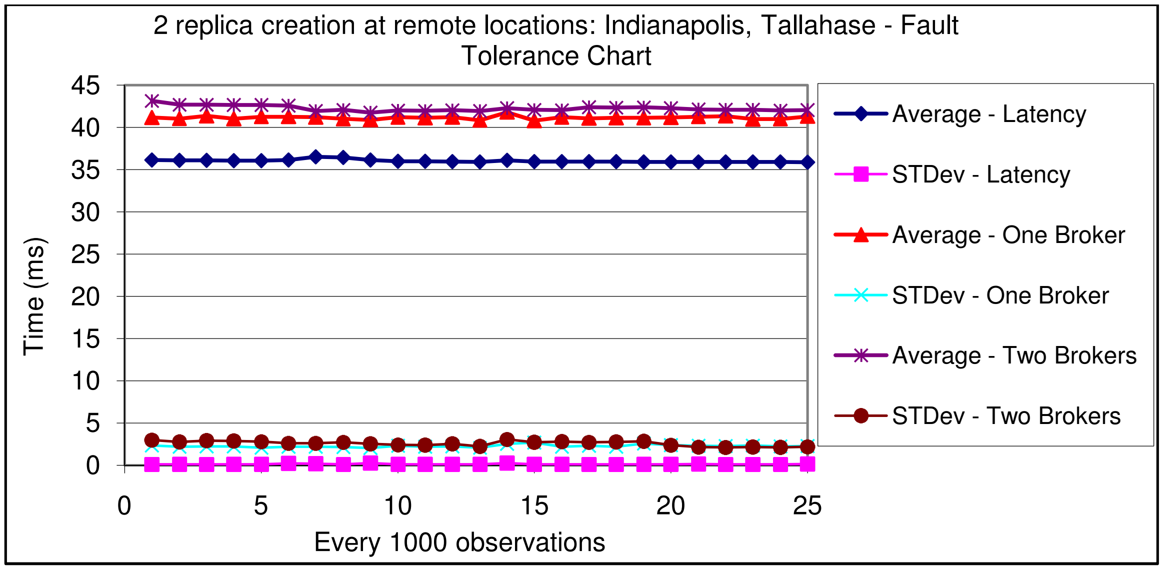


Figure 44 Fault Tolerance Experiment results when two replicas are created at two remote locations: Indianapolis, IN and Tallahase, FL. Each point in the graph corresponds to average of 1000 observations.

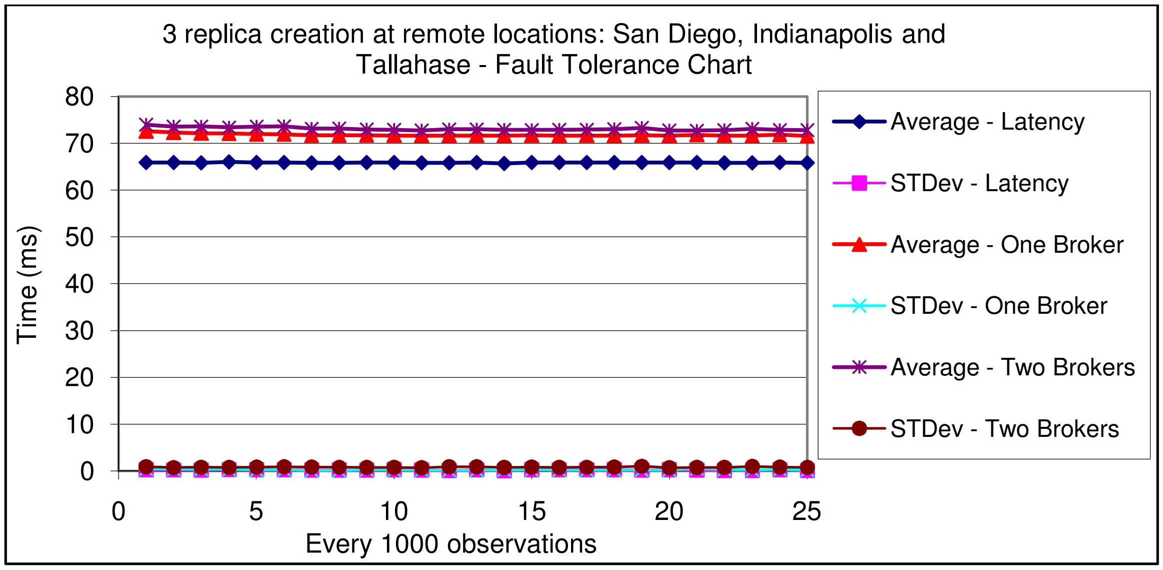


Figure 45 Fault Tolerance Experiment results when three replicas are created at three remote locations: Indianapolis, IN, Tallahase, FL and San Diego, CA. Each point in the graph corresponds to average of 1000 observations.

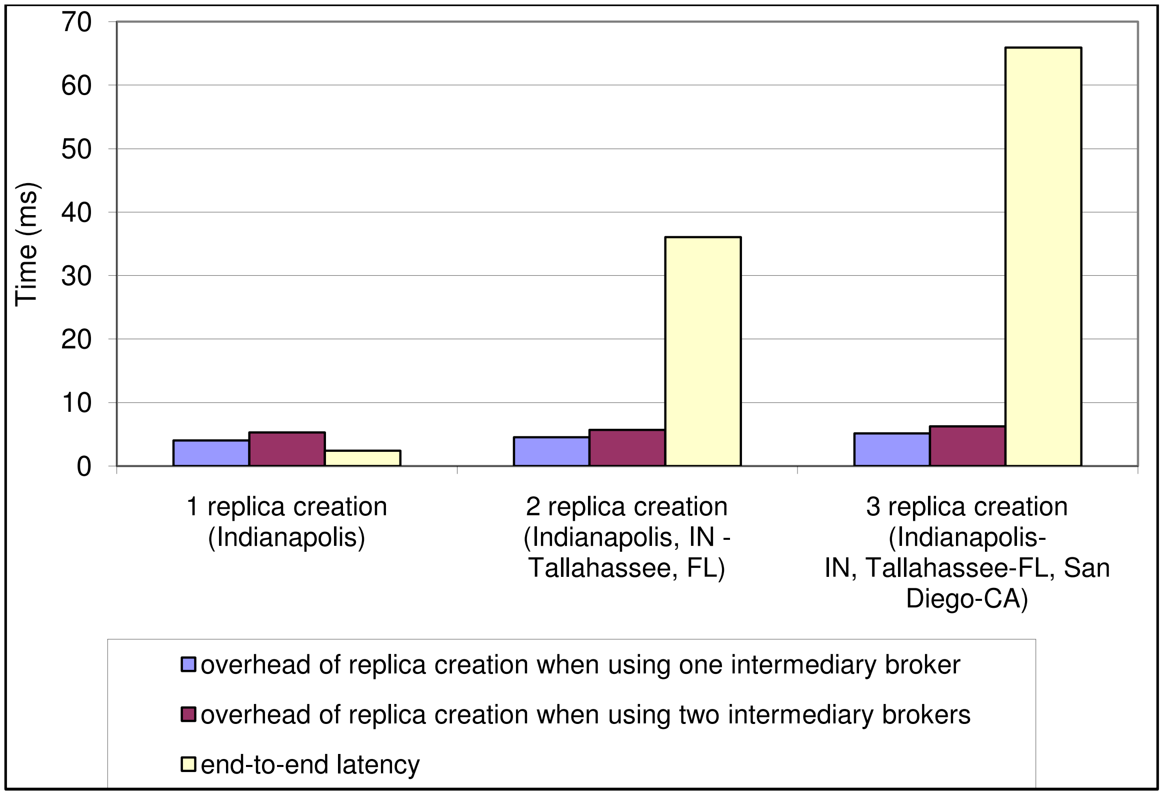


Figure 46 Time spent in various sub-activities of the replica-content creation scheme of the Hybrid Service.

	one broker	two brokers	end-to-end latency
1 replica (Indianapolis)	4.02	5.27	2.43
2 replicas (Indianapolis–Tallahassee)	4.54	5.67	36.05
3 replicas (Indianapolis–Tallahassee –San Diego)	5.13	6.24	65.90

Table 14 Statistics for Figure 46. Overhead of replica-content creation. Average timings in milliseconds.

Based on the results depicted in Figure 43, Figure 44, and Figure 45, we extract the processing time involved to provide fault-tolerance by utilizing publish-subscribe based messaging schemes. We depict the time spent in various sub-activities of replica creation in Figure 46 and list in Table 14. By analyzing the results, we observe that the system presents a stable performance over time for replica creation. We observe that the time required for one replica creation is only four milliseconds. The cost of replica creation time includes the Hybrid Service system processing overhead and overhead of using an intermediary broker as part of publish-subscribe system. We also observe that the time required for replica creation increases, as the number of replica copies increases. This is because; the system has to perform an additional unicast message for each additional replica creation. The time required for a unicast message is less than one millisecond. The results also indicated that, the overhead of replica-content creation increases only by 1.2 ms, when we use an additional intermediary broker.

6.7 Consistency Enforcement Experiment

The design of the consistency enforcement is similar to the distribution experiment depicted in Figure 35. In this experiment, our aim is to answer the following questions: a) What is the cost of consistency enforcement in terms of the time required to carry out updates at the primary-copy holder?, b) How does the system behavior change for continuous, uninterrupted update operations (for consistency enforcement)? To this end, we conducted two tests: The first test was conducted with one broker where the broker is located before the Hybrid Service instance in Bloomington, IN, while the second test was conducted with two broker nodes each sitting on the same machine before the Hybrid Service instances. In this experiment, we measured the time required to distribute an update request to the primary-copy holder of the context under consideration for consistency enforcement reasons.

6.7.1 Consistency Enforcement Experiment Results

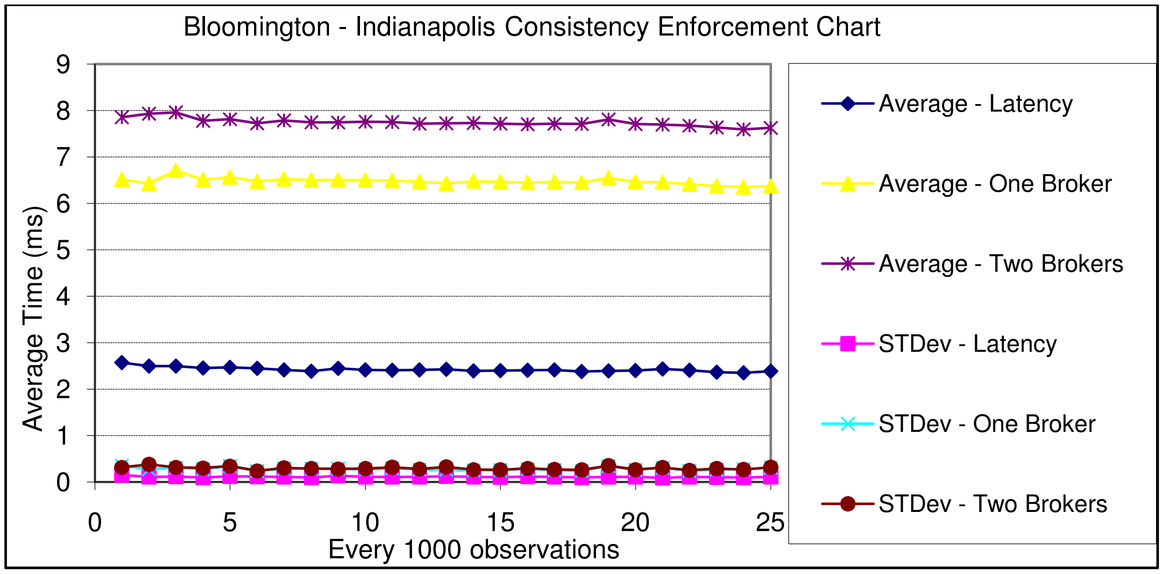


Figure 47 Consistency Enforcement Experiment Results when an update request (originated from Bloomington, IN) is carried out on the primary-copy holder located in Indianapolis, IN. Each point in the graph corresponds to average of 1000 observations.

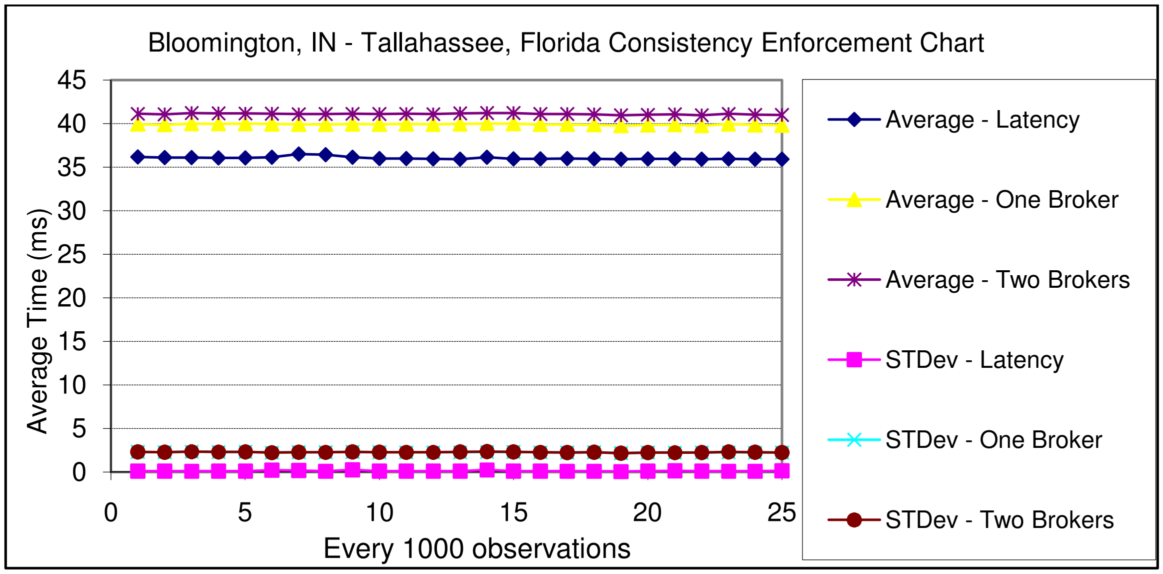


Figure 48 Consistency Enforcement experiment results when an update request (originated from Bloomington, IN) is carried out on the primary-copy holder located in Tallahassee, FL. Each point in the graph corresponds to average of 1000 observations.

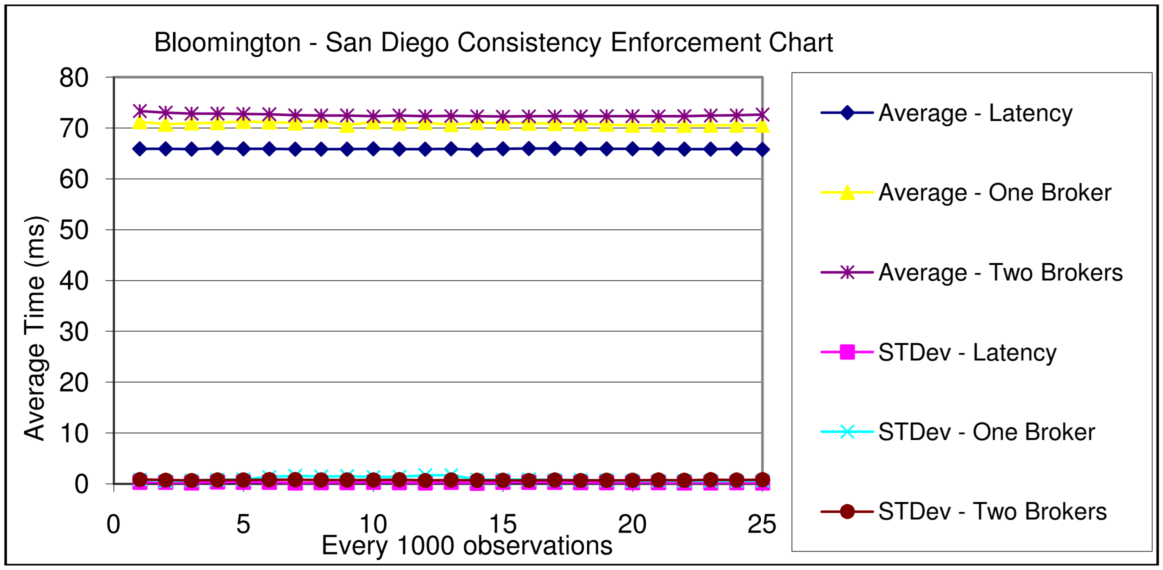


Figure 49 Consistency Enforcement Experiment Results when an update request (originated from Bloomington, IN) is carried out on the primary-copy holder located in San Diego, CA. Each point in the graph corresponds to average of 1000 observations.

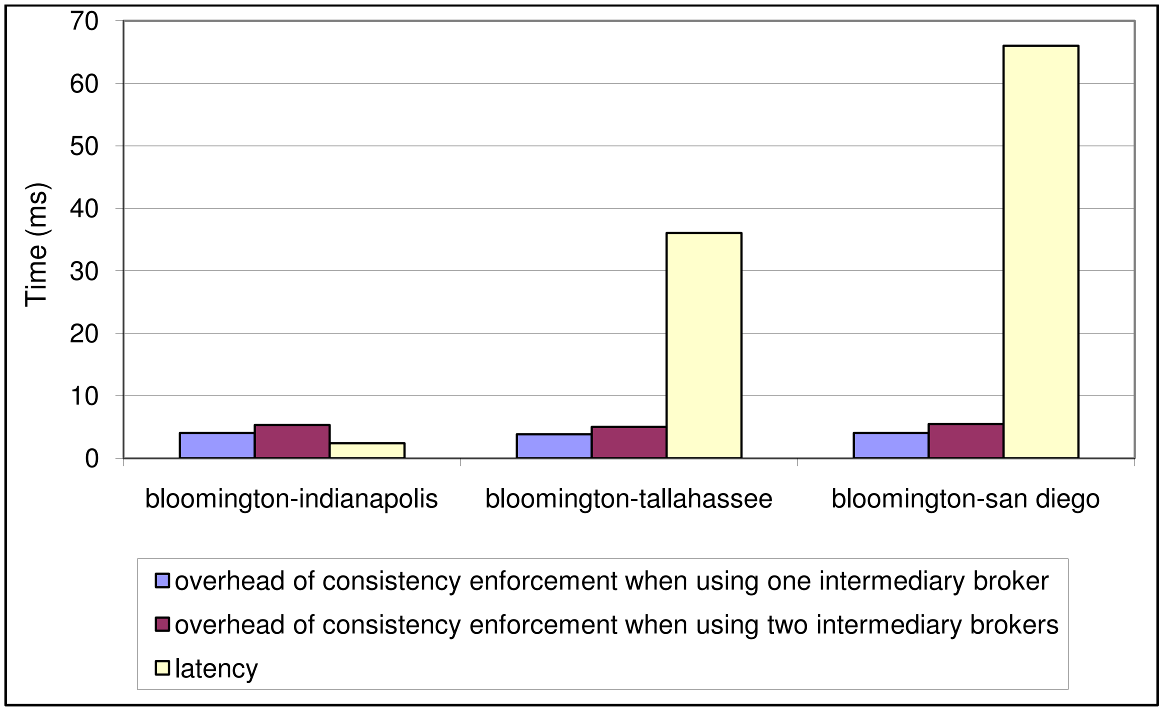


Figure 50 Time spent in various sub-activities of the Hybrid Service consistency enforcement scheme. The results analyze the overhead of distributing update requests to the primary-copy holder where the update requests take place for consistency enforcement reasons.

	one broker	two brokers	end-to-end latency
Bloomington – Indianapolis	4.05	5.32	2.42
Bloomington – Tallahassee	3.83	5.03	36.05
Bloomington – San Diego	4.07	5.49	66

Table 15 Statistics for Figure 50. Statistics for overhead of update distribution. Average timings in milliseconds.

Based on the results depicted in Figure 47, Figure 48 and Figure 49, we extract the processing time involved to provide consistency enforcement using publish-subscribe based messaging schemes. We depict the time spent in various sub-activities of distributing and carrying out the update request at the primary-copy holder in Figure 50 and list in Table 15. This cost of consistency enforcement includes the Hybrid Service system processing overhead (for distributing update request to primary-copy holder) and overhead of using an intermediary broker as part of publish-subscribe system. We observe that the time required for consistency enforcement does not change regardless of how Hybrid System instances are distributed. Similar to our results in the previous two experiments, we observe that the overhead of consistency enforcement increases only by 1.2 ms when we use an additional intermediary broker. By analyzing the results, we also observe that the system presents a stable performance over time for continuous consistency enforcement operations.

6.8 Summary

This chapter presented the performance evaluation of the Hybrid Service. Firstly, the evaluation indicated that metadata management systems could provide remarkable performance achievements by simply employing an in-memory storage mechanism, while preserving persistency of information. The results pointed out the trade-off between the persistency and performance. The results showed that performance is increased for a standard publish operation, when the back-up frequency is chosen as a small number. In other words, if the system uses bigger time interval for back-up, it performs better.

Secondly, it pointed out the trade-off between the scalability and performance. Based on the results, we discovered some threshold values for the maximum number of simultaneous publish or inquiry operations that can be performed on the system. For example, when the number of inquiry workload exceeds 800 simultaneous messages per second, the performance of the system starts dropping down. Therefore, the higher scalability, the lower the performance would be for a standard operation, when the workload of the system exceeds certain threshold values. The results also showed that the system is able scale to increasing message sizes and performs well.

Thirdly, it pointed that Hybrid Service presents stable behavior for access request distribution, replica creation and consistency enforcement over a high number continuous operations. The results indicated that, with our solution, the cost of achieving distribution, fault tolerance and consistency enforcement is in the order of milliseconds. We also observed that the cost of fault tolerance is higher than both the cost of distribution and the cost of consistency enforcement. This is because; there is an additional time required for performing additional unicast messages for higher fault-tolerance levels.

Fourthly, it pointed out that we can achieve performance optimization by employing dynamic replication technique in decentralized metadata management. The results indicated that the cost of repetitive access requests could be reduced by moving temporary copies of contexts to where they wanted.

Finally, it pointed out the trade-off between performance and fault-tolerance. Here the fault-tolerance is considered in terms of availability (i.e. degree of replication). The results indicated that the cost of replica-content creation increases, when the degree of fault-tolerance increased.

Chapter 7

Conclusion and Future Work

7.1 Thesis Summary

This thesis studied Grid Information Services to address metadata management requirements of application use domains described in Section 1.3.2. We determined the scope of this research by identifying the metadata management requirements of motivating application use domains. Section 1.3.1 discussed these requirements in details. We discussed the relevant work in Chapter 2. Having identified the requirements and reviewed the previous solutions, we proposed semantics and an architectural design for a Hybrid Grid Information System. We introduced the Hybrid Grid Information Service Architecture in Chapter 3. We discussed its semantics in Chapter 4 and explained its prototype implementation in Chapter 5. We introduced empirical evaluation of the system in Chapter 6.

Firstly, the proposed Hybrid Service architecture provides unification of Grid Information Services. It forms an add-on architecture that interacts with the local information systems and unifies them in a higher-level hybrid system. In other words, the Hybrid Service provides a unifying architecture where one can assemble metadata instances of different information services. To achieve this, the Hybrid System Architecture introduces various abstraction layers for uniform access interface and information resource management. Each information service has its own customized schema and communication protocol. The uniform access abstraction layer is implemented to support one to many communication protocols. The information resource management abstraction layer is implemented to manage one to many local schema implementations. In our prototype implementation, we have shown that the Hybrid Service is able to unify the two local information service implementations: WS-Context and Extended UDDI and support their communication protocols.

Secondly, the proposed Hybrid Service provides federation of information in Grid Information Services. This capability enables federation of Grid Information Services in metadata instances. To achieve this capability, the Hybrid Service requires a global schema integrating local information service schemas and user-provided mapping rules to provide mappings. To facilitate testing of this capability, we introduced a Unified Schema as a common communication platform and it's Query/Publish XML API as a shared common language. We have also introduced mapping rules as XSLT files between the Unified Schema and local information service schemas. The Hybrid Service performs transformations between instances of the Unified Schema and the local schemas based on the user-provided mapping rules. With this capability, we enable different Grid

Information Service implementations to interact with each other and share each other's metadata. Furthermore, with this approach, we provide the ability to issue integrated queries on the heterogeneous metadata space where metadata comes from different information service providers. This allows us 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 have shown an example of the federation capability, by introducing a Unified Schema integrating the three local information service schemas: extended UDDI, Glue and WS-Context. We have also introduced an integrated communication protocol that allows users to publish metadata instances into the heterogeneous metadata space. In our prototype implementation, the Hybrid Grid Information Service supported transformations between the Unified Schema and the two local information service schemas: WS-Context and extended UDDI, as we provided the implementations of these two information services.

Thirdly, the Hybrid Service is implemented as a high performance information system. With the Hybrid Service approach, we introduced an in-memory storage, which runs one layer above existing local information services. To achieve persistency of information, the Hybrid Service occasionally stores newly-inserted/updated metadata into appropriate local information service backend. To implement the in-memory storage capability, the proposed system utilizes an associative shared memory platform (by utilizing the JavaSpaces Specification).

Fourthly, the Hybrid Service is implemented as a decentralized system. To achieve decentralization, we utilized publish-subscribe based messaging schemes to provide interaction among the distributed instances of the Hybrid System. We utilized a topic based publish-subscribe messaging communication to implement fundamental aspects of decentralized information systems such as fault-tolerance, distribution, and consistency enforcement. To improve the overall performance of the system, we have also used performance optimization techniques such as dynamic migration/replication, which improves overall system performance by moving/replicating highly requested metadata to where they wanted.

Fifthly, we have implemented the WS-Context Service based on the WS-Context Specifications to provide an efficient mediator service supporting communication among services in dynamically assembled Grid/Web service collections. The proposed Hybrid System runs as an add-on architecture, one layer above the implementation of the WS-Context Service. The WS-Context Specification models interaction-dependent, session metadata as an external entity where more than two services can access/store highly dynamic shared metadata. It intends to manage the lifecycle of dynamic information within an activity. We termed an activity as "session" and dynamically generated information associated to it as "interaction-dependent metadata". In this thesis, to implement the WS-Context Service, we introduce semantics, which consists of data model and programming interface (see Sections 4.2). In the prototype implementation of WS-Context Service, we provided advanced query capabilities to support distributed state management and collaboration session management. Examples of these capabilities could be a) support for real-time replay and b) session-failure recovery.

Sixthly, we have implemented an extended version of existing UDDI Specification. This is an information service designed to address metadata management requirements of Geographical Information Systems, yet it also provides domain-independent advanced query capabilities. With this implementation, we introduce information model (see Section 4.3.1) and access interface (see Section 4.3.2). The extended UDDI information model includes entities, where additional metadata associated to a Web Service, can be stored. Its programming interface provides metadata-oriented publishing/discovery capabilities. The additional XML API set introduces various capabilities such as publishing additional metadata associated with service entries, posing metadata-oriented, geospatial, and domain-independent queries. The domain-independent search capability is a more general-purpose extension to the UDDI data model. It allows us to insert arbitrary XML metadata into the repository. This way, the metadata catalog may be searched using XPATH queries, a standard way for searching XML documents.

Finally, we have performed a set of experiments to evaluate the performance the Hybrid Service. We conducted a performance experiment (see Section 6.2) where the results showed that information services could provide significant performance achievements by employing an in-memory storage while preserving a certain level of persistency. We conducted scalability experiments (see Section 6.3) where the results indicated that the Hybrid Service is able to respond well to large number of concurrent requests without any error. This experiment also showed that the system performs well for increasing metadata sizes. We conducted a distribution experiment (see Section 6.4) where we investigated the performance and stability of our distribution methodology. The results indicated that the cost of providing request distribution is only few milliseconds and the system performance does not degrade for uninterrupted, continuous operations. We conducted an experiment to test if the dynamic replication mechanism works (see

Section 6.5). The results showed that the system is able to move/replicate highly requested metadata to where the requests are originated. We have also investigated the performance and stability of our methodologies for replica-content creation and consistency enforcement (see Sections 6.6 and 6.7). The results indicated that the processing cost of having fault-tolerance and enforcing consistency is only few milliseconds and the system presents a stable performance for continuous operations.

7.2 Answers to Research Questions

We answer the aforementioned research questions (see Section 1.2) based on our findings:

1) Can we implement a hybrid system architecture that unifies custom implementations of Grid Information Services to provide a common access interface to different kinds of service-metadata in Service Oriented Architectures?

The answer to this question is “yes”. We introduced a Hybrid Grid Information Service that is an add-on architecture above the existing grid information services. It presents abstraction layers for both metadata access and information-resource management. It is able to support one to many information service implementation backends and their communication protocols. It unifies their metadata under a higher-level structure. Chapter 3 overviewed the architecture, Chapter 4 discussed its abstract data models and Chapter 5 explained the prototype implementation of the architecture. This approach provided a uniform access interface to different kinds of service-metadata in Service Oriented Architectures.

2) How can we provide federation of information among the Grid Information Services, so that they can share/exchange metadata with each other? What is a common data model and communication protocol for such federation capability?

We observe that different Grid applications adopt customized implementations of Grid Information Services. These information services support different communication protocols and they are not interoperable with each other. This creates a challenge, as different Grid domains cannot share/exchange metadata and communicate with each other. To address this challenge, we built a federation capability integrated within the Hybrid Grid Information Service Architecture. To achieve this, we introduced a Unified Schema Specification by integrating the data models of Extended UDDI, WS-Context and Glue Specifications. We also introduced a shared communication protocol to achieve an integrated access to heterogeneous information space. The Hybrid Service allows users to provide their own mapping rules to map Unified Schema instances to the other local schema instances. It performs the transformations based on the user-provided mapping rules. Chapter 4 discussed the Unified Schema abstract data models and XML API that allowed us to create the federation capability among different Grid Information Services. Chapter 5 discussed the prototype and explained how the Hybrid Service achieves the federation capability in detail.

3) What is the efficient metadata access/storage strategy for such a hybrid system architecture that could speed up performance of existing Grid Information Services and that could provide persistency of information?

To meet the performance requirement of the research problem, the Hybrid Grid Information Service is designed as an in-memory storage, which runs one layer above the existing information services to improve their performance. To provide an in-memory storage, we utilized the TupleSpaces asynchronous communication paradigm (see Section 2.3). The TupleSpaces concept provides an associative lookup capability and is an appropriate model when there are multiple-writers sharing the data. All metadata accesses happen in memory to minimize average transaction execution time of the standard operations. In order to achieve persistency, we implemented a persistency management capability, which backs-up newly inserted/updated information into appropriate information service backend every so often. The experimental studies discussed in Sections 6.2 and 6.3 showed that the proposed methodology provides an efficient performance in metadata access/storage, while providing persistency of information at the same time.

4) What are the efficient request distribution, replica-content creation, and consistency enforcement strategies to achieve decentralized hybrid information system architecture? Can we implement these fundamental features of the decentralized system with publish-subscribe based messaging schemes?

To meet the fault-tolerance and performance requirements of the research problem, we implemented the Hybrid System as a decentralized information service with efficient distribution, replica-content creation, look-ahead caching and consistency enforcement schemes. To implement these fundamental issues of designing a decentralized replica hosting system, we use the topic-based publish-subscribe paradigm. We discuss our implementation methodology in Chapter 5 with detail. Chapter 6 discussed experiments investigating our approach. Based on our results, we have found that one can achieve efficient distribution, fault-tolerance and consistency enforcement capabilities by utilizing publish-subscribe based messaging schemes with negligible processing overheads.

5) How does the decentralized system behavior change for continuous operation?

By analyzing the results gathered from different experiments (see Sections 6.4, 6.6, and 6.7) evaluating the fundamental aspects of our replica hosting system, we observed that the system performance does not degrade because of continuous operations. Thus, we concluded that the system presented stability for continuous request distribution, replica-content creation and consistency enforcement operations.

6) How can we achieve a self-adopting decentralized information service architecture that can answer instantaneous client-demand changes?

By analyzing the metadata requirements of our application use domains, we observed that metadata might have volatile behavior and have changing user demands. To meet the dynamism requirement, we implemented the dynamic replication algorithm introduced by Rabinovich et al [104]. This approach provided a self-adopting capability into the system. This way the system captures the dynamic behavior both in metadata and network topology. The dynamic replication methodology replicates/migrates metadata to handle sudden bursts of client requests coming from unexpected remote locations. Section 5.6.7 discussed our dynamic replication approach and Section 6.5 discussed the experiments. The results showed that the dynamic replication works in decentralized metadata management architectures and provides performance optimization in metadata access.

7) Can we support communication among Grid/Web Services with efficient mediator information service methodologies?

Based on the performance results given in Sections 6.2 and 6.3, we have shown that communication among services could be achieved with efficient centralized metadata strategies (such as the WS-Context approach, see Section 2.1.2), with metadata coming from more than two services. Our performance results indicated that the processing overhead of metadata access and storage is very small (see Section 6.2). In contrast, point-to-point methodologies provide service conversation with metadata only from the two services that exchange information. Our approach also showed that, by employing the centralized approach, one could perform collective operations such as queries on subsets of all available metadata in service conversation. We have shown example of this with Hybrid Grid Information Service integrated with the WS-Context Service.

7.3 Future Research Directions

This thesis revisited distributed data management techniques to achieve integrated access to heterogeneous metadata in Grid Information Services. It introduced a Unified Schema (by integrating different information service schemas) and provided mappings between the Unified Schema and local schemas based on user-provided mapping rules. We plan to expand on this approach to be able to scale up large number of metadata sources. We will further research decentralized schema mapping strategies to express high-level queries over the local schemas without relying on a global Unified Schema.

An additional area that we intend to research that is needed to complete the system is an information security mechanism for the distributed Hybrid Service. This research should investigate the security concerns related to communication between network nodes and users, as well as security concerns related to authorization to deal with access control.

Appendix A: Supported XML API Sets

Supported XML API Index
A1. The WS-Context Service XML API Sets
A2. The Extended UDDI Service XML API Sets
A3. The Unified Information Service XML API Sets
A4. Hybrid Information Service Generic Web Service Interface

A.1. The WS-Context Service XML API Sets

The API Sets of the WS-Context XML Metadata service can be grouped as following: 1) **Inquiry**, 2) **Publication**, 3) **Security** and 4) **Proprietary** XML API Sets.

A.1.1. The WS-Context Service Inquiry XML API Set

We introduced various API calls representing inquiries that can be used to retrieve data from the WS-Context Service.

find_sessionService: The find session service API call is a functionalitiy of the WS-Context XML Metadata Service. It locates services matching the conditions specified in the query.

Syntax:

```
<find_sessionService [maxRows=”nn”] [listHead="0]>
        [<findQualifiers>]
        [<authInfo>]
        [<sessionKey>]
        [<name>]
        [<xpathExpression>]
        [<context>]
        [<lease>]
</find_ sessionService >
```

Attributes:

- *maxRows*: The optional integer value that allows the requesting program to limit the number of results returned.

- *listHead*: The optional integer value indicates which item should be returned as the head of the list first.

Arguments:

- *findQualifiers:* The optional collection of find Qualifier elements can be used to alter the behaviour of the search functionality.
- *authInfo*: The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.
- *sessionKey*: The session uuid_keys are used to specify one to many instances of a sessionEntity element in the hybrid service. If the sessionKeys are specified, only those services that are associated with these sessionKeys will be searched.
- *name:* This optional collection of string values represent one or more names given to session service entities. This argument is used together with an appropriate wildcard character specified in the findQualifiers. For instance, as the default wildcard is "exactMatch", if the name argument is specified, any serviceEntity matching the specified names will be searched.
- *xpathExpression:* This optional element is used if the query needed to be placed in in-memory storage.
- *context:* This optional argument defines a list of dynamic metadata (context) that are to be associated with a service instance. If the context is

specified, only those services that are associated with these contexts will be searched.

- *lease:* This optional argument defines a time period during which the requested list of web services are up and running.

Returns:

This API call returns a list of session service entities matching the query on success. In the event that no matches were located for the specified criteria, the service entity array structure returned will be empty. This signifies zero messages. If no arguments are passed a zero-match result set will be returned.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

find_session: The find_session API call is related with the WS-Context. It is used to find session entity elements.

Syntax:

```
<find_session [maxRows=”nn”] [listHead="0]>
        [<findQualifiers>]
        [<authInfo>]
        [<serviceKey>]
        [<name>]
        [<xpathExpression>]
        [<context>]
        [<lease>]
</find_ session >
```

Arguments:

- *findQualifiers:* The optional collection of find Qualifier elements can be used to alter the behaviour of the search functionality.

- *authInfo*: The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.
- *serviceKey*: This *uuid_key(s)* is used to specify a particular instance of a service element in the registered data. If the *serviceKey(s)* is specified, only those session entities that are associated with the given *serviceKey(s)* will be searched.
- *name:* This is an identifier given by the user to session entities. This argument is used together with an appropriate wildcard character specified in the findQualifiers. For instance, as the default wildcard is "exactMatch", if the identifier argument is specified, any session entity matching the specified identifier will be searched.
- *xpathExpression:* This optional element is used if the query needed to be placed in in-memory storage.
- *context:* This optional argument defines a list of dynamic metadata (context) that are to be associated with a service instance. If the context is specified, only those sessions that are associated with these contexts will be searched in the system.
- *lease:* This optional argument defines a time period during which the requested list of sessions are declared to be valid.

Returns:

This API call returns a list of session entities matching the query on success. In the event that no matches were located for the specified criteria, the session entity array structure returned will be empty. This signifies zero messages. If no arguments are passed a zero-match result set will be returned.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

find_context: The find_context API call is related with the WS-Context. It is used to find context entity elements.

Syntax:

```
<find_context [maxRows=”nn”] [listHead="0]>
        [<findQualifiers>]
        [<authInfo>]
        [<sessionKey>]
        [<serviceKey>]
        [<name>]
        [<xpathExpression>]
        [<lease>]
</find_ context >
```

Arguments:

- *findQualifiers:* The optional collection of find qualifier elements can be used to alter the behaviour of the search functionality.
- *authInfo*: The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.
- *sessionKey*: This *uuid_key(s)* is used to specify a particular instance of a session entity element. If the *sessionKey(s)* is specified, only those context elements that are associated with the given *sessionKey(s)* will be searched.

- *serviceKey*: This *uuid_key(s)* is used to specify a particular instance of a service element in the registered data. If the *serviceKey(s)* is specified, only those context entities that are associated with the given *serviceKey(s)* will be searched.
- *name:* This is an identifier given by the user to context entities. This argument is used together with an appropriate wildcard character specified in the findQualifiers. For instance, as the default wildcard is "exactMatch", if the identifier argument is specified, any context entity matching the specified identifier will be searched.
- *xpathExpression:* This optional element is used if the query needed to be placed in in-memory storage.
- *lease:* This optional argument defines a time period during which the requested list of contexts are declared to be valid.

Returns:

This API call returns a list of context entities matching the query on success. In the event that no matches were located for the specified criteria, the context entity array structure returned will be empty. This signifies zero messages. If no arguments are passed a zero-match result set will be returned.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

get_sessionServiceDetail: The get_sessionServiceDetail is related with the WS-Context. It returns the sessionService structure corresponding to specified serviceKey(s).

Syntax:

```
<get_serviceDetail >
        [<authInfo>]
        [<serviceKey>]
</ get_serviceDetail >
```

Arguments:

- *authInfo*: The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.
- *serviceKey*: This *uuid_key(s)* is used to specify a particular instance of a sessionService element. If the *serviceKey(s)* is specified, only those businessService elements that are associated with the given *serviceKey(s)* will be searched.

Returns:

This API call returns a sessionServiceDetail element on success. A sessionServiceDetail is an XML element, which contains an array of sessionService structures. In the event that no matches were located for the specified criteria, the sessionServiceDetail element will not contain any sessionService elements. If no arguments are passed a zero-match result set will be returned.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

get_sessionDetail: The get_sessionDetail API call is used to retrieve sessionEntity data structure corresponding to each of the session key values specified in the arguments. It is a functionality related with the WS-Context Schema.

Syntax:

```
<get_ sessionDetail >
        [<authInfo>]
        [<sessionKey>]
</ get_ sessionDetail >
```

Arguments:

- *authInfo:* The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.
- *sessionKey: This uuid_key(s)* is used to specify a particular instance of a sessionEntity element. If the *sessionKey(s)* is specified, only those sessionEntity structures that are associated with the given *sessionKey(s)* will be searched/retrieved.

Returns:

This API call returns a sessionDetail element on success. A sessionDetail is an XML element, which contains an array of sessionEntity structures. If no arguments are passed a zero-match result set will be returned.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

get_contextDetail: The get_contextDetail API call is used to retrieve the context structure corresponding to the context key values specified in the argument list. It is a functionality related with the WS-Context Schema.

Syntax:

```
<get_ contextDetail >
```

```
        [<authInfo>]
        [<contextKey>]
</ get_ contextDetail >
```

Arguments:

- *authInfo:* The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.
- *contextKey: This uuid_key(s)* is used to specify a particular instance of a context element. If the *contextKey (s)* is specified, only those context structures that are associated with the given *contextKey (s)* will be searched/retrieved.

Returns:

This API call returns a contextDetail element on success. A contextDetail is an XML element, which contains one to many context context elements, which are associated with the specified contextKey(s) in the arguments. If no arguments are passed a zero-match result set will be returned.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

get_contents: The get_contents API call is using syncrounous callback style communication for sending requests. It is defined by WS-Context Specification. The syntax of this call is exactly the same as the get_contextDetail(…) function. However, this function does not send anything directly in response to the request. Thus, the only difference between the get_contents(…) and the get_contextDetail(…) functions is that the former uses a synrounous call-back style communication while the latter utilizes RPC-style communication. The get_contents(…) function contains the call-back address of a ContextRespondant Service in passing arguments. The ContextRespondant Service is used to communicate with the system using syncrounous callback style functions. It allows the system to send results to a client who initiates a publication or inquiry callback style operations. Similar to the get_contextDetail(…), this function is also used to retrieve the context structure corresponding to the context key values specified in the argument list. It is a functionality related with the WS-Context Schema.

Syntax:

```
<get_contents >
        [<authInfo>]
        [<contextKey>]
</ get_contents >
```

Arguments:

- *authInfo:* The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.
- *contextKey: This uuid_key(s)* is used to specify a particular instance of a context element. If the *contextKey (s)* is specified, only those context structures that are associated with the given *contextKey (s)* will be searched/retrieved.

Returns:

This API call uses a synchronous call-back for communication. Thus, it does not return anything directly in response to the request. Instead all results are sent to a ContextRespondant Service using synrounous call-backs. In response to the get_contents(…), the “contents” function of the ContextRespondant Service is invoked. This is needed to return the details of a context (a contextDetail element). A contextDetail is an XML element, which contains the entire context under consideration.

Caveats:

If any error occurs in processing this API call, following two functions may be invoked on the ContextRespondant Service: a) unknownContextFault(): This message is sent to indicate the specified context could not be found for update, and b) generalFault(): This message is sent to indicate that some other error occurred during the execution of the function. These two functions are part of ContextRespondant Service which is defined by WS-Context Specifications, so not covered here.

A.1.2. The WS-Context Service Publish XML API Set

We introduce various extensions to XML API of the Context Manager of the WS-Context Specification to publish and update session-related metadata associated with services.

save_sessionService: The save session service API call is related with the WS-Context XML Metadata Service. It allows users to update or add one or more sessionService elements into the WS-Context XML Metadata Service.

Syntax:

```
<save_sessionService >
        [<authInfo>]
        [<sessionService>]
</ save_ sessionService >
```

Arguments:

- *authInfo:* The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.
- *sessionService:* This is a required argument which consists of one or more sessionService elements. A sessionService element contains a sessionKey, which is *uuid_key* used to specify the category under which the service is to be published, and a sessionKey, used to specify the session to which the service is being participated. If the serviceKey, an identifier used to specify the service, is passed with the sessionService element, then the system updates the entries associated with the serviceKey.

Returns:

This API call returns a sessionServiceDetail element, which contains the resulting sessionService structures after publication of new information.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

save_session: The save_session API call is used to add/update one or more session entities. It is related with the WS-Context XML Metadata Service.

Syntax:

```
<save_session >
        [<authInfo>]
        [<sessionEntity>]
</ save_session >
```

Arguments:

- *authInfo:* The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.
- *sessionEntity:* This is a required argument which consists of one or more aforementioned sessionEntity elements.

Behaviour:

If the sessionKey, an identifier used to specify the sessionEntity, is passed within the sessionEntity element, then the system updates the entries associated with the specified sessionKey.

Returns:

This API call returns a sessionDetail element, which contains the information after publication/update operation, takes place for the affected sessionEntity elements.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

save_context: The save_context API call is used to add/update on or more context (dynamic metadata) entities into the service. It is related with the WS-Context XML Metadata Service.

Syntax:

```
<save_context >
        [<authInfo>]
        [<context>]
</ save_context >
```

Arguments:

- *authInfo:* The authInfo element is an optional argument containing an authentication token. If there is a required restricted access, then this argument is passed.
- *context:* This is a required argument which defines a list of dynamic metadata (context) that are to be associated with a sessionEntity or sessionService instance. In order to do an update operation, a context structure may be first obtained by using get_contextDetail operation.

Behaviour:

If a contextKey, an identifier used to specify a particular context, is passed within the context element, then this is a signal for the system that the corresponding context exists in the system. So, the system updates the entries associated with the specified contextKey.

If a contextKey is passed with an empty value, then the system behave as if the dynamic context under consideration is being inserted for the first time. So, the system generates a unique identifier corresponding to this context and new entries are inserted associated with the newly generated contextKey.

Returns:

This API call returns a contextDetail element on success. A contextDetail contains the final version of context(s) after publication or update operation.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

set_contents: The set_contents API call is using syncrounous callback style for sending requests. It is defined by WS-Context Specification. The syntax of this call is the same as save_context function. However, this function does not send anything directly in response to the request. Similar to the save_context(…), this function is also used to add/update one or more context (interaction-dependent metadata) entities into the service.

Syntax:

```
<set_contents >
        [<authInfo>]
        [<context>]
</set_contents >
```

Arguments:

- *authInfo:* The authInfo element is an optional argument containing an authentication token. If there is a required restricted access, then this argument is passed.
- *context:* This is a required argument which defines a list of dynamic metadata (context) that are to be associated with a sessionEntity or sessionService instance.

Behaviour:

The behaviour of this function is almost the same as the aforementioned save_context function. The only difference between the set_contents(…) and the save_context(…) is that the former uses a synrounous call-back style communication while the latter utilizes RPC-style communication. Thus, the set_contents(…) function contains the call-back address of a ContextRespondant Service in its arguments. The ContextRespondant Service is used to communicate with the system using syncrounous call-back style functions. It allows the hybrid service to sent results to a client who initiates a publication or inquiry callback style operations.

Returns:

This API call is a synchronous callback function. Thus, it does not return anything directly in response to the request. Instead, all results are sent to the ContextRespondant Service using synrounous callbacks. In response to the set_contents(…) function, the contentsSet function of the ContextRespondant Service is invoked. This is needed to indicate that the contents of the context have been stored/updated successfully.

Caveats:

If any error occurs in processing this API call, following two functions may be invoked on the ContextRespondant Service: a) unknownContextFault(): This message is sent to indicate the specified context could not be found for update, and b) generalFault(): This message is sent to indicate that some other error occurred during the execution of the function. These two functions are part of ContextRespondant Service, which is defined by WS-Context Specifications, so not covered here.

delete_sessionService: The delete_sessionService API call is related with the WS-Context XML Metadata Service. It is used to delete existing session service entities associated with the specified service_Key(s) from the system.

Syntax:

```
<delete_service >
        [<authInfo>]
        [<serviceKey>]
</ delete_service >
```

Arguments:

- *authInfo:* The authInfo element is an optional argument containing an authentication token. If there is a required restricted access, then this argument is passed.
- *serviceKey*: This is a required argument and used to specify a particular instance of a service element. When this argument is passed, one or more service entitles associated with the specified *serviceKey(s)* will be deleted.

Returns:

When a successful deletion operation is executed a success message is returned to the client.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

delete_session: The delete_session API call is a functionality related with the WS-Context Schema and used to delete one or more sessionEntity structures from the system.

Syntax:

```
<delete_session >
        [<authInfo>]
        [<sessionKey>]
</ delete_session >
```

Arguments:

- *authInfo:* The authInfo element is an optional argument containing an authentication token. If there is a required restricted access, then this argument is passed.

- *sessionKey*: This is a required argument and used to specify a particular instance of a sessionEntity element. When this argument is passed, one or more sessionEntity structures associated with the specified *sessionKey(s)* will be deleted.

Returns:

When a successful deletion operation is executed a success message is returned to the client.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

delete_context: The delete_context API call is a functionality related with the WS-Context Schema and used to delete one or more context structures from the system.

Syntax:

```
<delete_context >
        [<authInfo>]
        [<contextKey>]
</ delete_context >
```

Arguments:

- *authInfo:* The authInfo element is an optional argument containing an authentication token. If there is a required restricted access, then this argument is passed.
- *contextKey*: This is a required argument and used to specify a particular instance of a context element. When this argument is passed, one or more context elements associated with the specified *contextKey(s)* will be deleted.

Returns:

When a successful deletion operation is executed a success message is returned to the client.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

A.1.3. The WS-Context Service Security and Proprietary API

The WS-Context XML Metadata Service adopts the semantics for the Security XML API (get_authToken, discard_authToken) and Proprietary XML API (save_publisher, get_publisherDetail, find_publisher and delete_publisher) from existing UDDI Specifications [8]. We implement these semantics to provide find/add/modify/delete operations on the publisher list, i.e., authorized users of the system. These XML APIs include the following function calls. We must note that the WS-Context Securiy API is implemented for the centralized WS-Context metadata registry approach. Based on this implementation, the centralized service requires an authentication token to restrict who can perform inquiry/publish operation. The authorization token is obtained from the service at the beginning of client-server interaction. In this scnerio, 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.

A.1.3.1. Security XML API

get_authToken: The get_authToken API call is used to request an authentication token as an "authInfo" (authentication information) element from the service. Both publication and inquiry API set includes authentication information in their input arguments.

discard_authToken: The discard_authToken API call is used to inform hybrid WS-Context service that a particular authentication token is no longer required and should be considered as invalid.

A.1.3.2. Proprietary XML API

find_publisher: The find_publisher API call is used to find publishers registered with the system matching the conditions specified in the arguments.

save_publisher: The save_publisher API call is used to add or update information about a publisher.

delete_publisher: The delete_publisher API call is used to delete information about a publisher with a given publisherID.

get_publisherDetail: The get_publisherDetail API call is used to retrieve detailed information regarding one or more publishers with given publisherID(s).

A.2 Extended UDDI XML Service XML API Set

The API Sets of the extended UDDI XML Metadata service can be grouped as following: 1) **Inquiry** and 2) **Publish**.

A.2.1 Extended UDDI Service Inquiry XML API Set

We introduced various API calls representing inquiries that can be used to retrieve data from the Extended UDDI XML Metadata Service.

find_service: This API is a functionality related with the extended UDDI. The find service API call locates services matching the conditions specified in the query. Each find_service query enables metadata oriented query capabilities. This capability concern with static and rarely changing attributes of services(s).

Syntax:

```
<find_service [maxRows=”nn”] [listHead="0]>
        [<findQualifiers>]
        [<authInfo>]
        [<businessKey>]
        [<names>]
        [<xpathExpression>]
        [<categoryBag>]
        [<serviceAttribute>]
        [<lease>]
        [<tModelBag>]
</find_service>
```

Attributes:

- *maxRows*: The optional integer value that allows the requesting program to limit the number of results returned.
- *listHead*: The optional integer value indicates which item should be returned as the head of the list first.

Arguments:

- *findQualifiers:* The optional collection of find Qualifier elements can be used to alter the behaviour of the search functionality.
- *authInfo*: The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.

- *businessKey*: This *uuid_key* is used to specify a particular instance of a businessEntity element in the registered data. If the businessKey is specified, only those services that are associated with the *businessKey* will be searched.
- *name:* This optional collection of string values represent one or more names given to businessService entities. This argument is used together with an appropriate wildcard character specified in the findQualifiers. For instance, as the default wildcard is “exactMatch”, if the name argument is specified, any serviceEntity matching the specified names will be searched.
- *xpathExpression:* This optional element is used if the query needed to be placed in in-memory storage.
- *categoryBag:* This optional argument is a list of category references. When this argument is used the returned list of services will contain element matching all the categories passed (by default logical operation AND is set).
- *serviceAttribute:* This optional argument defines a list of static metadata (service attributes) that are to be associated with a service instance. If the service attribute is specified, only those services that are associated with these service attribute will be searched in the registry.
- *lease:* This optional argument defines a time period during which the requested list of web services are up and running.

- *tModelBag:* This optional argument is a collection of uuid_key elements specifiying that search results are to be limited to those services that expose themselves with a technical fingerprint that match.

Returns:

This API call returns a list of businessService entities matching the query. In the event that no matches were located for the specified criteria, the service entity array structure returned will be empty. This signifies zero messages. If no arguments are passed a zero-match result set will be returned.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

find_serviceAttribute: This API is a functionality related with the extended UDDI. It is used to locate serviceAttribute elements matching the query.

Syntax:

```
<find_serviceAttribute [maxRows=”nn”] [listHead="0]>
        [<findQualifiers>]
        [<authInfo>]
        [<serviceKey>]
        [<xpathExpression>]
        [<categoryBag>]
        [<lease>]
</find_ serviceAttribute >
```

Arguments:

- *findQualifiers:* The optional collection of find qualifier elements can be used to alter the behaviour of the search functionality.
- *authInfo*: The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.

- *serviceKey*: This *uuid_key* is used to specify a particular instance of a businessService element in the registered data. If the *serviceKey* is specified, only those serviceAttributes that are associated with the *serviceKey* will be searched.
- *xpathExpression:* This optional element is used if the query needed to be placed in in-memory storage.
- *categoryBag:* This optional argument is a list of category references. When this argument is used the returned list of service attributes will be matching all the categories passed (by default logical operation AND is set).
- *lease:* This optional argument defines a time period during which the requested list of service attributes are valid.

Returns:

This API call returns a list of service attribute entities matching the query. In the event that no matches were located for the specified criteria, the service attribute entity array structure returned will be empty. This signifies zero messages. If no arguments are passed a zero-match result set will be returned.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

get_serviceDetail: This API is a functionality related with the extended UDDI. The get_serviceDetail returns the extended businessService structure corresponding to specified serviceKey(s). The system returns a businessService structure, which contains interaction-independent (static) information.

Syntax:

```
<get_serviceDetail >
        [<authInfo>]
        [<serviceKey>]
</ get_serviceDetail >
```

Arguments:

- *authInfo*: The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.
- *serviceKey*: This *uuid_key(s)* is used to specify a particular instance of a businessService element. If the *serviceKey(s)* is specified, only those businessService elements that are associated with the given *serviceKey(s)* will be searched.

Returns:

This API call returns a serviceDetail element on success. A serviceDetail is an XML element, which contains an array of businessService structures. In the event that no matches were located for the specified criteria, the serviceDetail element will not contain any businessService elements. If no arguments are passed a zero-match result set will be returned.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

get_serviceAttributeDetail: The get_serviceAttributeDetail is a functionality of the extended UDDI. It is used to retrieve semi-static metadata associated to a unique identifier. The system retrieves the requested serviceAttribute structures corresponding to specified attributeKey(s) and returns the results as an array of serviceAttributes within an element called serviceAttributeDetail. The result is returned back to the querying user.

Syntax:

```
<get_ serviceAttributeDetail >
        [<authInfo>]
        [<attributeKey>]
</ get_ serviceAttributeDetail >
```

Arguments:

- *authInfo:* The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.
- *attributeKey: This uuid_key(s)* is used to specify a particular instance of a serviceAttribute element. If the *attributeKey(s)* is specified, only those serviceAttribute elements that are associated with the given *attributeKey(s)* will be searched/retrieved.

Returns:

This API call returns a serviceAttributeDetail element on success. A serviceAttributeDetail is an XML element, which contains an array of serviceAttribute structures. If no arguments are passed a zero-match result set will be returned.

A.2.2 The Extended UDDI Service Publish XML API Set

We introduced various API calls representing inquiries that can be used to retrieve data from the Extended UDDI XML Metadata Service.

save_service: The save_service API is related with the extended UDDI Schema. The save service API call allows users to update or add one or more businessService elements into the extended UDDI XML Metadata Service. It uses the same syntax with the out-of-box UDDI save service.

Syntax:

```
<save_service >
        [<authInfo>]
        [<businessService>]
</ save_service >
```

Arguments:

- *authInfo:* The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.
- *businessService:* This is a required argument which consists of one or more businessService elements.

Returns:

This API call returns a serviceDetail element, which contains the resulting businessService structures after publication of new information.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

save_serviceAttribute: The save_serviceAttribute API is related with the extended UDDI Schema. Here, the system handles the serviceAttribute publication operation and returns a serviceAttributeDetail element.

Syntax:

```
<save_serviceAttribute >
        [<authInfo>]
        [<serviceAttribute>]
```

</ save_serviceAttribute >

Arguments:

- *authInfo:* The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.
- *serviceAttribute:* This is a required argument which consists of one or more serviceAttribute elements. A serviceAttribute is a static metadata. It contains a serviceKey, which is *uuid_key* used to specify the service entry under which this metadata is to be published. If the attributeKey, an identifier used to specify the serviceAttribute, is passed within the serviceAttribute element, then the system updates the entries associated with the attributeKey.

Returns:

This API call returns a serviceAttributeDetail element on success. A serviceAttributeDetail contains the final version of serviceAttribute(s) after publication or update operation.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

delete_service: The delete_service API is related with the extended UDDI Schema and used to delete existing service entities associated with the specified service_Key(s) from the system. This API call adopts the syntax from the out-of-box UDDI delete function for interoperability.

Syntax:

```
<delete_service >
        [<authInfo>]
        [<serviceKey>]
</ delete_service >
```

Arguments:

- *authInfo:* The authInfo element is an optional argument containing an authentication token. If there is a required restricted access, then this argument is passed.
- *serviceKey*: This is a required argument and used to specify a particular instance of a businessService element. When this argument is passed, one or more service entitles associated with the specified *serviceKey(s)* will be deleted.

Returns:

When a successful deletion operation is executed, an empty message is returned to the client.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

delete_serviceAttribute: The delete_serviceAttribute is a functionality related with the extended UDDI Schema. It is used to delete service attribute metadata associated to services.

Syntax:

```
<delete_serviceAttribute >
        [<authInfo>]
        [<attributeKey>]
```

```
</ delete_serviceAttribute >
```

Arguments:

- *authInfo:* The authInfo element is an optional argument containing an authentication token. If there is a required restricted access, then this argument is passed.
- *attributeKey*: This is a required argument and used to specify a particular instance of a serviceAttribute element. When this argument is passed, one or more static-metadata entries associated with the specified *attributeKey(s)* will be deleted.

Returns:

When a successful deletion operation is executed, an empty message is returned to the client.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

A.3. The Unified Schema XML API Set

The Hybrid Information Service introduces a Unified Schema and XML API to provide a common information model and query/publish syntax for both interaction-dependent and interaction-independent metadata spaces. The Unified Schema XML API set can be grouped as following: 1) **Inquiry** and 2) **Publish**.

A.3.1. The Unified Schema Inquiry XML API:

We introduced various API calls representing inquiries that can be used to retrieve data from the Hybrid Service using the Unified Schema XML API.

find_service: The find service API call is a functionality of the Unified Schema. It locates services matching the conditions specified in the query.

Syntax:

```
<find_service [maxRows=”nn”] [listHead="0]>
        [<authInfo>]
        [<name>]
        [<xpathExpression>]
</find_ service >
```

Attributes:

- *maxRows*: The optional integer value that allows the requesting program to limit the number of results returned.
- *listHead*: The optional integer value indicates which item should be returned as the head of the list first.

Arguments:

- *authInfo*: The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.
- *name:* This optional string value represents a name given to service entity by the user.
- *xpathExpression:* This optional element is used if the query needed to be placed in in-memory storage.

Returns:

This API call returns a list of service entities matching the query on success. In the event that no matches were located for the specified criteria, the service entity array structure returned will be empty. This signifies zero messages. If no arguments are passed a zero-match result set will be returned.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

find_business: The find business API call is a functionalitiy of the Unified Schema. It locates business entity instances matching the conditions specified in the query.

Syntax:

```
<find_business [maxRows=”nn”] [listHead="0]>
        [<authInfo>]
        [<name>]
        [<xpathExpression>]
</find_business >
```

Attributes:

- *maxRows*: The optional integer value that allows the requesting program to limit the number of results returned.
- *listHead*: The optional integer value indicates which item should be returned as the head of the list first.

Arguments:

- *authInfo*: The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.

- *name:* This optional string value represents a name given to business entity by the user.
- *xpathExpression:* This optional element is used if the query needed to be placed in in-memory storage.

Returns:

This API call returns a list of business entities matching the query on success. In the event that no matches were located for the specified criteria, the business entity array structure returned will be empty. This signifies zero messages. If no arguments are passed a zero-match result set will be returned.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

find_session: The find_session API call is related with the Unified Schema. It is used to find session entity elements.

Syntax:

```
<find_session [maxRows=”nn”] [listHead="0]>
        [<authInfo>]
        [<name>]
        [<xpathExpression>]
</find_session >
```

Arguments:

- *authInfo*: The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.

- *name:* This optional string value represent a name given to session entity by the user.
- *xpathExpression:* This optional element is used if the query needed to be placed in in-memory storage.

Returns:

This API call returns a list of session entities matching the query on success. In the event that no matches were located for the specified criteria, the session entity array structure returned will be empty. This signifies zero messages. If no arguments are passed a zero-match result set will be returned.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

find_metadata: The find_metadata API call is related with the Unified Schema. It is used to find metadata entity elements.

Syntax:

```
<find_metadata [maxRows=”nn”] [listHead="0]>
        [<authInfo>]
        [<name>]
        [<xpathExpression>]
</find_ metadata >
```

Arguments:

- *authInfo*: The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.

- *name:* This optional string value represent a name given to session entity by the user.
- *xpathExpression:* This optional element is used if the query needed to be placed in in-memory storage.

Returns:

This API call returns a list of metadata entities matching the query on success. In the event that no matches were located for the specified criteria, the metadata entity array structure returned will be empty. This signifies zero messages. If no arguments are passed a zero-match result set will be returned.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

get_serviceDetail: The get_serviceDetail is a Unified Schema API call, which returns the service structure of the Unified Schema corresponding to specified serviceKey(s).

Syntax:

```
<get_serviceDetail >
        [<authInfo>]
        [<serviceKey>]
</ get_serviceDetail >
```

Arguments:

- *authInfo*: The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.

- *serviceKey*: This *uuid_key(s)* is used to specify a particular instance of a service element. If the *serviceKey(s)* is specified, only those service elements that are associated with the given *serviceKey(s)* will be searched.

Returns:

This API call returns a serviceDetail element on success. A serviceDetail is an XML element, which contains an array of service structures. In the event that no matches were located for the specified criteria, the serviceDetail element will not contain any service elements. If no arguments are passed a zero-match result set will be returned.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

get_sessionDetail: The get_sessionDetail API call is used to retrieve sessionEntity data structure corresponding to each of the session key values specified in the arguments. It is a functionality related with the Unified Schema.

Syntax:

```
<get_sessionDetail >
        [<authInfo>]
        [<sessionKey>]
</ get_sessionDetail >
```

Arguments:

- *authInfo:* The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.
- *sessionKey:* This *uuid_key(s)* is used to specify a particular instance of a sessionEntity element. If the *sessionKey(s)* is specified, only those

sessionEntity structures that are associated with the given *sessionKey(s)* will be searched/retrieved.

Returns:

This API call returns a sessionDetail element on success. A sessionDetail is an XML element, which contains an array of sessionEntity structures. If no arguments are passed, a zero-match result set will be returned.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

get_businessDetail: The get_businessDetail API call is used to retrieve businessEntity data structure corresponding to each of the business key values specified in the arguments. It is a functionality related with the Unified Schema.

Syntax:

```
<get_businessDetail >
        [<authInfo>]
        [<businessKey>]
</ get_businessDetail >
```

Arguments:

- *authInfo:* The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.
- *businessKey:* This *uuid_key(s)* is used to specify a particular instance of a businessEntity element. If the *businessKey(s)* is specified, only those businessEntity structures that are associated with the given *businessKey(s)* will be searched/retrieved.

Returns:

This API call returns a businessDetail element on success. A businessDetail is an XML element, which contains an array of businessEntity structures. If no arguments are passed, a zero-match result set will be returned.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

get_metadataDetail: The get_metadataDetail API call is used to retrieve the metadata structure corresponding to the metadata key values specified in the argument list. It is a functionality related with the Unified Schema.

Syntax:

```
<get_ metadataDetail >
        [<authInfo>]
        [<metadataKey>]
</ get_ metadataDetail >
```

Arguments:

- *authInfo:* The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.
- *metadataKey: This uuid_key(s)* is used to specify a particular instance of a metadata element. If the *metadataKey (s)* is specified, only those metadata structures that are associated with the given *metadataKey (s)* will be searched/retrieved.

Returns:

This API call returns a metadataDetail element on success. A metadataDetail is an XML element, which contains metadata elements, which are associated with the specified metadataKey(s) in the arguments. If no arguments are passed a zero-match result set will be returned.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

A.3.2. The Unified Schema Publish XML API:

save_service: The save service API call is a hybrid function which allows the users to update or add one or more service elements into the Hybrid Service using the Unified Schema XML API.

Syntax:

```
<save_service >
        [<authInfo>]
        [<service>]
</ save_service >
```

Arguments:

- *authInfo:* The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.
- *service:* This is a required argument which consists of one or more service elements.

Returns:

This API call returns a serviceDetail element, which contains the resulting service structures after publication of new information.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

save_business: The save business API call is a Unified Schema function which allows the users to update or add one or more business elements into the Hybrid Service using the Unified Schema XML API.

Syntax:

```
<save_business>
        [<authInfo>]
        [<businessEntity>]
</ save_business>
```

Arguments:

- *authInfo:* The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.
- *businessEntity:* This is a required argument which consists of one or more business entity elements.

Returns:

This API call returns a businessDetail element, which contains the resulting business structures after publication of new information.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

save_session: The save_session API call is used to add/update one or more session entities into the Hybrid Service. It is a functionality of the Unified Schema.

Syntax:

```
<save_session>
        [<authInfo>]
        [<sessionEntity>]
</ save_session>
```

Arguments:

- *authInfo:* The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.
- *sessionEntity:* This is a required argument which consists of one or more aforementioned sessionEntity elements.

Behaviour:

If the sessionKey, an identifier used to specify the sessionEntity, is passed within the sessionEntity element, then the system updates the entries associated with the specified sessionKey.

Returns:

This API call returns a sessionDetail element, which contains the information after publication/update operation, takes place for the affected sessionEntity elements.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

save_metadata: The save_metadata API call is used to add/update on or more metadata entities into the Hybrid Service. It is a functionality of the Unified Schema.

Syntax:

```
<save_metadata >
```

```
        [<authInfo>]
        [<metadata>]
</ save_metadata >
```

Arguments:

- *authInfo:* The authInfo element is an optional argument containing an authentication token. If there is a required restricted access, then this argument is passed.
- *metadata:* This is a required argument which defines a list of metadata that are to be associated with a session, service, or site.

Behaviour:

If a metadataKey, an identifier used to specify a particular metadata, is passed within the metadata element, then this is a signal for the system that the corresponding metadata exists in the system. So, the system updates the entries associated with the specified metadata Key.

If a metadata Key is passed with an empty value, then the system behave as if the metadata under consideration is being inserted for the first time. So, the system generates a unique identifier corresponding to this metadata and new entries are inserted associated with the newly generated metadata key.

Returns:

This API call returns a metadataDetail element on success. A metadataDetail contains the final version of metadata after publication or update operation.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

delete_service: The delete_service API call is a Unified Schema function, which is used to delete existing service entities associated with the specified service_Key(s) from the system.

Syntax:

```
<delete_service >
        [<authInfo>]
        [<serviceKey>]
</ delete_service >
```

Arguments:

- *authInfo:* The authInfo element is an optional argument containing an authentication token. If there is a required restricted access, then this argument is passed.
- *serviceKey*: This is a required argument and used to specify a particular instance of a service element. When this argument is passed, one or more service entitles associated with the specified *serviceKey(s)* will be deleted.

Returns:

When a successful deletion operation is executed a success message is returned to the client.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

delete_business: The delete_business API call is a Unified Schema function, which is used to delete existing business entities associated with the specified business_Key(s) from the system.

Syntax:

```
<delete_business>
        [<authInfo>]
        [<businessKey>]
</delete_business>
```

Arguments:

- *authInfo:* The authInfo element is an optional argument containing an authentication token. If there is a required restricted access, then this argument is passed.
- *businessKey*: This is a required argument and used to specify a particular instance of a business element. When this argument is passed, one or more service entitles associated with the specified *businessKey(s)* will be deleted.

Returns:

When a successful deletion operation is executed a success message is returned to the client.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

delete_session: The delete_session API call is a functionality related with the Unified Information Service Schema and used to delete one or more sessionEntity structures from the system.

Syntax:

```
<delete_session>
        [<authInfo>]
        [<sessionKey>]
</ delete_session>
```

Arguments:

- *authInfo:* The authInfo element is an optional argument containing an authentication token. If there is a required restricted access, then this argument is passed.
- *sessionKey*: This is a required argument and used to specify a particular instance of a sessionEntity element. When this argument is passed, one or more sessionEntity structures associated with the specified *sessionKey(s)* will be deleted.

Returns:

When a successful deletion operation is executed a success message is returned to the client.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

delete_metadata: The delete_metadata API call is a functionality related with the Unified Schema and used to delete one or more metadata structures from the system.

Syntax:

```
<delete_metadata >
        [<authInfo>]
        [<metadataKey>]
</ delete_ metadata >
```

Arguments:

- *authInfo:* The authInfo element is an optional argument containing an authentication token. If there is a required restricted access, then this argument is passed.

- *metadataKey*: This is a required argument and used to specify a particular instance of a metadata element. When this argument is passed, one or more metadata elements associated with the specified *metadataKey(s)* will be deleted.

Returns:

When a successful deletion operation is executed a success message is returned to the client.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

A.4. Hybrid Information Service Web Service Interface

In our prototype implementation, we have shown that the Hybrid Grid Information Service can support two widely used standards: UDDI and WS-Context Specifications. We have implemented a Hybrid Grid Information Service query/publish abstraction layer where one can use the XML API of different information services without changing the uniform access interface.

hybrid_function: The hybrid_function service API call allows users 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 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. Thre result is returned to user in XML format.

Syntax:

```
<hybrid_function>
        [<authInfo>]
        [<specName>]
        [<functionName>]
        [<requestXML>]
</hybrid_function>
```

Arguments:

- *authInfo*: The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.
- *specName:* This argument indicates the name of the schema which is under consideration.
- *functionName*: This argument indicates the name of the function.
- *requestXML*: This argument indicates the request in XML format based on the schema under consideration.

Returns:

This API call returns the result of the requested function in XML format based on the schema specified in the hybrid_function.

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 a hybrid function which allows the users to update/add one or more schema entity elements into the Hybrid Grid Information Service. This API is carried out on the JavaSpaces based in-memory storage. 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 as java objects into the JavaSpaces. Here, each schema entity object is stored associated with a unique identifier generated for the new publish request.

Syntax:

```
<save_schemaEntity>
        [<authInfo>]
        [<lease>]
        [<schema_Name>]
        [<schema_FunctionName>]
        [<schema_RequestXML>]
</ save_schemaEntity>
```

Arguments:

- *authInfo*: The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.
- *lease:* This optional argument defines a time period during which the requested list of web services are up and running.
- *schema_Name*: This argument is used to specify the schema under consideration.
- *schema_FunctionName*: This argument is used to identify the schema specific method which will be executed over the instance of the schema under consideration.
- *schema_RequextXML*: This argument is used to specify the actual publish function XML document, which is generated for saving an instance of an entity of the schema under consideration.

Returns:

This API call returns a schemaEntityDetail element, which contains the resulting schema entity structures after publication of new information.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

get_schemaEntityDetail: The get_schemaEntityDetail is a hybrid API call, which 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. This function is carried out on the JavaSpaces. 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, retrives the correct entity associated with the key from JavaSpaces. Finally, the system sends the result to the user.

Syntax:

```
<get_schemaEntityDetail>
        [<authInfo>]
        [<schema_Name>]
        [<schema_FunctionName>]
        [<schema_RequestXML>]
</get_schemaEntityDetail>
```

Arguments:

- *authInfo*: The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.
- *schema_Name*: This argument is used to specify the schema under consideration.

- *schema_FunctionName*: This argument is used to identify the schema specific method which will be executed over the instance of the schema under consideration.
- *schema_RequextXML*: This argument is used to specify the XML which is generated for retrieving an instance of an entity of the schema under consideration.

Returns:

This API call returns a schemaEntityDetail element, which contains the schema entity structure corresponding to a key.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

delete_schemaEntity: The delete_schemaEntity is a hybrid API call, which is used to delete an instance of any schema entities of a given Specification. The delete_schemaEntity API call is a hybrid function, which is used to delete existing service entities associated with the specified key(s) from the system. It is carried out on the JavaSpaces. 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. Finally, the system sends a success message whether the deletion is completed successfully.

Syntax:

```
<delete_schemaEntity>
        [<authInfo>]
        [<schema_Name>]
        [<schema_FunctionName>]
        [<schema_RequestXML>]
</delete_schemaEntity>
```

Arguments:

- *authInfo*: The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.
- *schema_Name*: This argument is used to specify the schema under consideration.
- *schema_FunctionName*: This argument is used to identify the schema specific method which will be executed over the instance of the schema under consideration.
- *schema_RequextXML*: This argument is used to specify the XML which is generated for deleting an instance of an entity of the schema under consideration.

Returns:

This API call returns a success element which contains a boolean variable indicating whether the deletion compleled with success.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

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 heteregenous tuple space where there exists tuples belong to different schemas. It is carried out on the JavaSpaces. 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.

Syntax:

```
<find_schemaEntity>
        [<authInfo>]
        [<entity_Type>]
        [<schema_Name>]
        [<schema_FunctionName>]
        [<schema_RequestXML>]
</find_schemaEntity>
```

Arguments:

- *authInfo*: The optional argument is an element containing an authentication token. If there is a required restricted access, then this argument is passed.
- *entity_Type*: This argument is used to specify the type of the entity which is being searched.
- *schema_Name*: This argument is used to specify the schema under consideration.
- *schema_FunctionName*: This argument is used to identify the schema specific method which will be executed over the instance of the schema under consideration.
- *schema_RequextXML*: This argument is used to specify the XML which is generated for finding an instance of an entity of the schema under specification.

Returns:

This API call returns a schemaEntityDetail element, which contains the schema entity structures matching the query.

Caveats:

If any error occurs in processing this API call, a dispositionReport element will be returned to the caller within a SOAP fault.

Appendix B: Example XML Metadata Documents

B.1. Sample 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.2. Sample UDDI XML metadata

```
<?xml version="1.0" encoding="UTF-8"?>
<uddi:businessService
        xmlns:uddi="http://uddi.services.axis.cgl/uddi_schema
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <serviceKey>12114460-B4B6-11DA-A1DD-C2341CB5D80D</serviceKey>
        <businessKey>7115B940-A95E-11DA-B940-CB4E3E38D62F</businessKey>
        <uddi:name>
        <value>Sample Service</value>
        </uddi:name>
        <uddi:description>
                <value>Service Description</value>
        </uddi:description>
        <value>String</value>
        <uddi:bindingTemplates>
        <uddi:bindingTemplate>
        <bindingKey>129679F0-B4B6-11DA-A1DD-E719F6E12358</bindingKey>
        <serviceKey>12114460-B4B6-11DA-A1DD-C2341CB5D80D</serviceKey>
        <uddi:accessPoint>
        <value>http://gf7.ucs.indiana.edu:8092/wfs-streaming-service/services/wfs</value>
        <useType>research</useType>
        </uddi:accessPoint>
        </uddi:bindingTemplate>
        </uddi:bindingTemplates>
        <uddi:categoryBag>
        <uddi:keyedReference>
                <uddi:tModelKey>6D712AF0-4ADA-11DA-BC65-C767C07EBBEA</uddi:tModelKey>
                <uddi:keyName>ServiceCategory</uddi:keyName>
                <uddi:keyValue>GIS-WFS</uddi:keyValue>
        </uddi:keyedReference>
        <uddi:categoryBag>
</uddi:businessService>
```

B.3. Sample 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>
```

References

1. Aydin, G., et al. SERVOGrid Complexity Computational Environments (CCE) Integrated Performance Analysis. in Grid Computing, 2005. The 6th IEEE/ACM International Workshop on. 2005: IEEE.
2. Wu, W., et al. A Web-services based conference control framework for heterogenous A/V collaboration. in 7th IASTED International Conference on INTERNET AND MULTIMEDIA SYSTEMS AND APPLICATIONS IMSA 2003 August 13-15, 2003 Honolulu, Hawaii, USA. 2003.
3. Wu, W., et al., Service Oriented Architecture for VoIP conferencing. International Journal of Communication Systems, 2006.
4. Fox, G., Grids of Grids of Simple Services. CISE Magazine July/August 2004.
5. B. Plale, P.D., and G. Von Laszewski. , Key Concepts and Services of a Grid Information Service. In Proceedings of the 15th International Conference on Parallel and Distributed Computing Systems (PDCS 2002), , 2002.
6. M. Gerndt, R.W., Z. Balaton, G. Gombás, P. Kacsuk, Zs. Németh, N. Podhorszki, H-L. Truong, T. Fahringer, M. Bubak, E. Laure, T. Margalef, Performance Tools for the Grid: State of the Art and Future. 2004, Shaker Verlag.
7. Zanikolas, S., Sakellariou, R., A Taxonomy of Grid Monitoring Systems. . Future Generation Computer Systems, 21(1), 2005: p. pp. 163--188.
8. 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.
9. LANL, Los Alamos National Laboratory, The Interdependent Energy Infrastructure Simulation System (IEISS) project, web site is available at http://www.lanl.gov/orgs/d/d4/interdepend.
10. Open Geospatial Consortium Inc., OpenGIS Web Map Service (WMS) Specification available at http://www.opengeospatial.org/standards/wms. 2006.
11. Open Geospatial Consortium Inc., OpenGIS Web Feature Service (WFS) Specification available at http://www.opengeospatial.org/standards/wfs. 2006.
12. Nacar, M., et al. VLab: Collaborative Grid Services and Portals to Support Computational Material Science. in GCE'05 Workshop on Grid Computing. Environments http://pipeline0.acel.sdsu.edu/mtgs/gce05 at SC05 Seattle, WA. 2005. 2005.
13. Oh, S. and G. Fox, HHFR: A new architecture for Mobile Web Services: Principles and Implementations, Technical Report. 2005, Community Grids Lab., Indiana University: Bloomington.
14. Oh, S., et al. Optimized communication using the SOAP infoset for mobile multimedia collaboration applications. in Collaborative Technologies and Systems, 2005. Proceedings of the 2005 International Symposium on. 2005.
15. Aktas, M., G. Fox, and M. Pierce. Managing Dynamic Metadata as Context. in Istanbul International Computational Science and Engineering Conference (ICCSE2005 http://www.iccse.org/) June 2005. 2005.
16. Aktas, M.S., G. Fox, and M. Pierce, An Architecture for Supporting Information in Dynamically Assembled Semantic Grids, Technical Report. 2005, Community Grids Lab., Indiana University.

17. Aktas, M.S., G. Fox, and M. Pierce. Information Services for Dynamically Assembled Semantic Grids. in Proceedings of 1st International Conference http://kg.ict.ac.cn/SKG2005/ on SKG2005 Semantics, Knowledge and Grid Beijing China November 27-29 2005. 2005.
18. Fox, G., Aktas, M., Aydin, G., Bulut, H., Pallickara, S., Pierce, M., Sayar, A., Wu, W., Zhai, G., Real Time Streaming Data Grid Applications, (IEEE) Proceedings of TIWDC 2005 CNIT Tyrrhenian International Workshop on Digital Communications July 4-6 2005. (To appear) in a book chapter in "Distributed Cooperative Laboratories: Networking, Instrumentation and Measurements" by F.Davoli, S. Palazzo, S. Zappatore, Eds., Springer,Norwell, MA, 2006, pp. 253-267.
19. Pallickara, S., Fox, G., Aktas, M., Gadgil, H., Yildiz, B., Oh, S., Patel, S., Pierce, M., Yemme, D., A Retrospective on the Development of Web Service Specifications in (To appear) in Securing Web Services: Practical Usage of Standards and Specifications. 2006, Edited by Dr. Periorellis Panos (University of Newcastle Upon Tyne) and published by Idea Group.
20. Aktas, M.S., Aydin, Galip, Fox, Geoffrey F., Gadgil, Harshawardhan, Pierce, Marlon, Sayar, Ahmet, Information Services for Grid/Web Service Oriented Architecture (SOA) Based Geospatial Applications, Technical Report. 2005, Community Grids Lab., Indiana University.
21. Aktas, M., et al. Web Service Information Systems and Applications. in GGF-16 Global Grid Forum Semantic Grid Workshop 2006. Athens, Greece.
22. Aktas, M.S., et al. Implementing Geographical Information System Grid Services to Support Computational Geophysics in a Service-Oriented Environment. in NASA Earth-Sun System Technology Conference http://esto.nasa.gov/conferences/estc2005/index.html University of Maryland, Adelphi, Maryland, June 28 - 30, 2005. All material is online for paper , presentation http://www.esto.nasa.gov/conferences/estc2005/Presentations/a6p2.pdf , and abstract http://www.esto.nasa.gov/conferences/estc2005/author.html. 2005.
23. Aydin, G., Sayar, A., Gadgil, H., Aktas, M., Fox, G., Ko, S., Bulut, H., Pierce, M. , Building and Applying Geographical Information System Grids, (To appear) in a special issue of Concurrency and Computation: Practice and Experience, Wiley.
24. 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.
25. Aktas, M.S., Fox, Geoffrey C., Pierce, Marlon E., Fault Tolerant High Performance Information Services for Dynamic Collections of Grid and Web Services. The International Journal of Grid Computing: Theory, Methods and Applications, Future Generation of Computer Systems (FGCS) Special issue from 1st International Conference on SKG2005 Semantics, Knowledge and Grid Beijing China November 27-29 2005, 2005.
26. 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.
27. Oh, S., Aktas, Mehmet S., Fox, Geoffrey C., Pierce, Marlon, Architecture for High-Performance Web Service Communications Using an Information Service. World

Scientific and Engineering Academy and Society Transactions on Information Science and Applications 2006.

28. Oh, S., Aktas, Mehmet S., Pierce, Marlon, Fox, Geoffrey C. Optimizing Web Service Messaging Performance Using a Context Store for Static Data. in Invited paper for 5th WSEAS Int.Conf. on TELECOMMUNICATIONS and INFORMATICS (TELE-INFO '06). 2006. Istanbul, Turkey.
29. Fox, G., et al., Algorithms and the Grid. Computing and Visualization in Science, 2006.
30. Fox, G., Aktas, M., Aydin, G., Donnellan, A., Gadgil, H., Granat, R., Pallickara, S., Parker, J., Pierce, M., Oh, S., Rundle, J., Sayar, A., Scharber, M. . Building Sensor Filter Grids: Information Architecture for the Data Deluge in the IEEE Proceedings of 1st International Conference on SKG2005 Semantics, Knowledge and Grid 2005. Beijing China
31. W3C Web Service Architecture Document, available at http://www.w3.org/TR/ws-arch/, 2003.
32. OGC, The Open Geospatial Consortiom (OGC), web site available at http://www.opengis.org.
33. Object_Management_Group, Catalog of Corba/IIOP specifications, available at http://www.omg.org/technology/documents/corbaservices_spec_catalog.htm.
34. SIG, B., Blootooth Specification available at http://bluetooth.com.
35. Ken Arnold, A.W., Byran O'Sullivan, Robert Scheifler, and Jim Waldo, The JINI Specification. 1999: Addison-Wesley, Reading, MA.
36. The_Salutation_Consortium_Inc., Salutation architecture specification (part 1), version 2.1 edition available at http://www.salutation.org. 1999.
37. Foster, I., Kesselman, C., Nick, Jeffrey M., Tuecke, S. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. in Open Grid Service Infrastructure WG, Global Grid Forum. 2002.
38. Chen, H., An Intelligent Broker Architecture for Context-Aware Systems. 2003, University of Maryland: Baltimore County.
39. Wu, D., Parsia, B., Sirin, E., Hendler, J., Nau, D. Automating DAML-S Web Services Composition Using SHOP2. in In 2nd International Semantic Web Conference (ISWC). 2003. Florida, USA.
40. Chakraborty, D., Perich, D., Avancha, S., Joshi, A. DReggie: A Smart Service Discovery Technique for E-Commerce Applications. in In Workshop in conjunction with 20th Symposium on Reliable Distributed Systems. 2001.
41. Aktas, M.S., et al. A Web based Conversational Case-Based Recommender System for Ontology aided Metadata Discovery. in GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04). 2004: IEEE.
42. Klyne, G., Carroll, J., Resource Description Framework (RDF): Concepts and Abstract Syntax. Latest version available at http://www.w3.org/TR/rdf-concepts/. 2004.
43. The_OWL_Service_Coalition, OWL-S:Semantic Markup for Web Services available at http://www.daml.org/services/owl-s/1.0/owl-s.html. 2003.
44. McGuinness, D.L., Harmelen, F. , OWL Web Ontology Language Overview. Latest version available at http://www.w3.org/TR/owl-features/. 2004.

45. Miles, S., Papay, J., Dialani, V., Luck, M., Decker, K., Payne, T., and Moreau, L. Personalized Grid Service Discovery. in Nineteenth Annual UK Performance Engineering Workshop (UKPEW'03). 2003. University of Warwick, Coventry, England.
46. Paolucci, M., Kawamura, T., Payne, T. R. and Sycara, K. Importing the Semantic Web in UDDI. in In Proceedings of Web Services, E-Business and Semantic Web Workshop, CAiSE 2002. , pages pp. 225-236. 2002. Toronto, Canada.
47. Guttman, E., Perkins, C., Veizades, J., Service Location Protocol, RFC 2165, available at http://rfc.net/rfc2165.html. 1997.
48. Milojicic, D.S., et al. , Peer-to-Peer Computing, in HP Labs Technical Report HPL-2002-57. 2002, HP Labs.
49. Fletcher, G., Sheth, H., Borner, K. Unstructured Peer-to-Peer Networks: Topological Properties and Search Performance. in the 3rd Int. Workshop on Agents and Peer-to-Peer Computing (AP2PC), at AAMAS 2004. 2004. New York City: Springer LNCS 3601, pp. 14-27.
50. MDS4, Monitoring & Discovery System (MDS4), web site is available at http://www.globus.org-/toolkit/mds.
51. A. Cooke, A.G., L. Ma, W. Nutt, J. Magowan, P. Taylor, R. Byrom, L. Field, S. Hicks, and J. Leake. , R-GMA: An Information Integration System for Grid Monitoring. Proceedings of the 11th International Conference on Cooperative Information Systems, 2003.
52. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S. A Scalable Content-Addressable Network. in Proc. ACM SIGCOMM, pp 161-172. 2001.
53. Stoica, I., Morris, R., Liben-Nowell, D., Karger, D. R., Kaashoek, M. F., Dabek, F., Balakrishnan, H. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. in IEEE/ACM Trans. on Networking. 2001.
54. Ripeanu, M., Foster, I. Mapping the Gnutella Network: Macroscopic Properties of Large Scale Peer-to-Peer Systems. in In 1st International Workshop on Peer to Peer Systems. 2002.
55. Lv, Q., Cao, P., Cohen, E., Li, K., Shenker, S. Search and Replication in Unstructured Peer to Peer Networks. in In 16th ACM International Concerence on SuperComputing. 2002. New York, USA.
56. T. Ozsu, P.V., Principles of Distributed Database Systems. 2nd Edition, Prentice Hall, 1999.
57. 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.
58. S. Helal, N.D., and C. Lee. Konark-A Service Discovery and Delivery Protocol for Ad-Hoc Networks. in In Third IEEE Conference on Wireless Communications Network (WCNC). March 2003. New Orleans, USA.
59. Marin-Perianu, R., Hartel, P., Scholten, J. A Classification of Service Discovery Protocols. in Technical Report TR-CTIT-05-25 Centre for Telematics and Information Technology, University of Twente, Enschede. ISSN 1381-3625 2005.
60. R. Hermann, D.H., M. Moser, M. Nidd, C. Rohner, A. Schade, DEAPspace--Transient ad hoc networking of pervasive devices. Computer Networks 2001. Volume 35 p. pp 411-428.

61. Tang, D., Chang, D., Tanaka, K., Baker, M., Resource Discovery in Ad-Hoc Networks, in CSL-TR-98-769. 1998, Stanford University.
62. Beatty, J., Kakivaya, G., Kemp, D., Kuehnel, T., Lovering, B., Roe, B., St. John, C., Schlimmer, J., Simonnet, G., Walter, D., Weast, J., Yarmosh, Y., and Yendluri, P. , Web Services Dynamic Discovery (WS-Discovery) available from http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-discovery.pdf. 2004.
63. Booth, D., et al. W3C Web Service Architecture Document, available at http://www.w3.org/TR/ws-arch/. 2003.
64. Open_GIS_Consortium_Inc., OWS-1 Registry Service available at http://www.opengeospatial.org/docs/01-024r1.pdf. 2002.
65. Open Geospatial Consortium Inc., OpenGIS Catalog Specification available at http://portal.opengeospatial.org/files/index.php?artifact_id=901.
66. OASIS_ebXML_Registry_Technical_Committee, OASIS/ebXML Registry Information Model v2.0 Approved Committee Specification available at http://www.oasis-open.org/committees/regrep/documents/2.0/specs/ebRIM.pdf.
67. ebXML, ebXML Registy Specification, web site available at http://www.ebxml.org.
68. Zhao, P., Chen, A., Liu, Y., Di, L., Yang, W., Li, P. Grid metadata catalog service-based OGC web registry service. in Proceedings of the 12th annual ACM international workshop on Geographic information systems. 2004. Washington DC, USA
69. Singh, G., Bharathi, S., Chervenak, A., Deelman, E., Kesselman, C., Manohar, M., Patil, S. and Pearlman. L. . A Metadata Catalog Service for Data Intensive Applications. in SC'03. November 15-21, 2003. Phoenix, Arizona, USA.
70. Web Service Interoperability (WS-I) Organization, web site is available at http://www.ws-i.org.
71. 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.
72. 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.
73. Sycline, Sycline Inc., web site available at http://www.synclineinc.com.
74. Galdos, Galdos Inc., web site available at http://www.galdosinc.com.
75. Dialani, V., UDDI-M Version 1.0 API Specification. 2002, University of Southampton – UK. 02.: Southampton.
76. Miles, S., Papay, J., Payne, T., Decker, K., Moreau, L. Towards a Protocol for the Attachment of Semantic Descriptions to Grid Services. in In The Second European across Grids Conference. 2004. Nicosia, Cyprus.
77. 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.
78. GRIMOIRES - UDDI compliant Web Service registry with metadata annotation extension, availble at http://sourceforge.net/projects/grimoires.
79. MyGrid - UK e-Science project, available at http://www.mygrid.org.uk.

80. Czajkowski, K., Ferguson, D., Foster, I., Frey, J., Graham, S., Sedukhin, I., Snelling, D., Tuecke, S., Vambenepe, W., The WS-Resource Framework, available at http://www.globus.org/wsrf/specs/ws-wsrf.pdf. 2004.
81. Ballinger, K., et al., The Web Services Metadata Exchange http://specs.xmlsoap.org/ws/2004/09/mex/WS-MetadataExchange.pdf. 2004.
82. 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.
83. Bunting, B., Chapman, M., Hurley, O., Little M,, Mischinkinky, J., Newcomer, E., Webber, J., and Swenson, K., Web Service Composite Application Framework (WS-CAF) Ver 1.0. http://developers.sun.com/techtopics/webservices/wscaf/primer.pdf. July 2003.
84. Bunting, B., Chapman, M., Hurley, O., Little M,, Mischinkinky, J., Newcomer, E., Web-ber, J., and Swenson, K, WS-Coordination Framework (WS-CF) ver 1.0 http://www.arjuna.com/library/specs/ws_caf_1-0/WS-CF.pdf. July 2003.
85. Bunting, B., Chapman, M., Hurley, O., Little M,, Mischinkinky, J., Newcomer, E., Web-ber, J., and Swenson, K. , WS-Transaction Management (WS-TXM) ver 1.0 http://www.arjuna.com/library/specs/ws_caf_1-0/WS-TXM.pdf. July 2003.
86. Pallickara, S., H. Gadgil, and G. Fox. On the Discovery of Topics in Distributed Publish/Subscribe systems. in Proceedings of the IEEE/ACM GRID 2005 Workshop, http://pat.jpl.nasa.gov/public/grid2005/ pp 25-32. Seattle, WA. 2005.
87. Happner, M., Burridge, R., Sharma, R., Java Message Service Specification available at http://java.sun.com/products/jms. 2000, Sun Microsystems.
88. Box, D., Cabrera, L., Crithchley, C., Curbera, F., Ferguson, D., Geller, A., Graham, S., Hull, D., Kakivaya, G., Lewis, A., Lovering, B., Mihic, M., Niblett, P., Orchard, D., Saiyed, J., Samdarshi, S., Schlimmer, J., Sedukhin, I., Shewchunk, J., Smith, B., Weerawarana, S., Wortendyke, D. , Web Service Eventing available at http://ftpna2.bea.com/pub/downloads/WS-Eventing.pdf. 2004, Microsoft, IBM & BEA.
89. 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.
90. Pallickara, S., et al., A Framework for Secure End-to-End Delivery of Messages in Publish/Subscribe Systems. 2005.
91. Pallickara, S., et al., Support for High Performance Real-time Collaboration within the NaradaBrokering Substrate. 2005.
92. Fox, G., S. Pallickara, and X. Rao. A scaleable event infrastructure for peer to peer grids. in JGI '02: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande. 2002: ACM.
93. Fox, G. and S. Pallickara, Deploying the NaradaBrokering substrate in aiding efficient web and grid service interactions. Proceedings of the IEEE, 2005. 93(3): p. 564-577.
94. Carriero, N., Gelernter, D., Linda in Context. Commun. ACM, 32(4): 444-458, 1989.
95. Sun_Microsystems, JavaSpaces Specification Revision 1.0, 1999 available at http://www.sun.com/jini/specs/js.ps.

96. Wyckoff, P., Lehman, T. J., McLaughry, S., T Spaces. IBM Systems Journal, 1998. 37(3): p. 454-474.
97. 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.
98. 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.
99. 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.
100. Sun_Microsystems, Sun Microsystems Inc., web site available at http://www.sun.com.
101. 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.
102. Sivasubramanian, S., Szymaniak, M., Pierre, G., Steen, M., Replication for Web Hosting Systems. ACM Computing Surveys, 36(3):291--334, 2004.
103. Tanenbaum, A., Van Steen, M., Distributed Systems Principles and Paradigms. 2002. Cited in page 326.
104. Rabinovich, M., Rabinovich, I., Rajaraman, R., Aggarwal, A. A Dynamic Object Replication and Migration Protocol for an Internet Hosting Service. in Proc. 19th Int'l Conf. Distributed Computing Systems. 1999.
105. Dilley, J., Maggs, B., Parikh, J., Prokop, H., Sitaraman, R., and Weihl, B., Globally distributed content delivery. IEEE Internet Computing, 2002: p. pp 50-58.
106. Rodriguez, P., Sibal, S. , SPREAD: Scalable Platform for Reliable and Efficient Automated Distribution. Computer Networks, 2000. vol. 33, nos. 1-6: p. pp. 33-49.
107. Rabinovich, M., Aggarwal, A. RaDaR: A Scalable Architecture for a Global Web Hosting Service. in WWW8. May 1999.
108. Aktas, M.S., et al., Information Services for Grid/Web Service Oriented Architecture (SOA) Based Geospatial Applications, Technical Report. 2005.
109. Rabinovich, M. Issues in Web Content Replication. in Bulleting of the IEEE Computer Society Technical Committee on Data Engineering. 1998.
110. Pallickara, S., Fox, G. NaradaBrokering: A Distributed Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids. in Proceedings of ACM/IFIP/USENIX International Middleware Conference Middleware-2003. 2003. Rio Janeiro, Brazil.
111. GLUE Schema Collaboration. The GLUE Schema homepage. http://infnforge.cnaf.infn.it/glueinfomodel/.
112. Rahm, E., Bernstein, P., A survey of approaches to automatic schema matching. . VLDB Journal10 (2001) 334-350.
113. Bernstein, P., Applying model management to classical meta data problems In Proc. CIDR (2003) 209-220.
114. Sun_Microsystems, Java(TM) 2 SDK, Standard Edition, version 1.4.2 available at http://java.sun.com/javase.
115. Axis, Apache Axis Web Service Development Platform, web site is available at http://ws.apache.org/axis/.

116. Community_Grids_Lab, NaradaBrokering Messaging Infrastructure available at http://www.naradabrokering.org/software.htm.
117. Viens, S., Cutright, A., Saldhana, A. , jUDDI, A free, open source and java implementation the Universal Description, Discovery, and Integration (UDDI) specification for Web Services http://ws.apache.org/juddi/.
118. Aktas, M.S., Fault Tolerant High Performance Information Service - FTHPIS - Hybrid WS-Context Service web site, available at http://www.opengrids.org/wscontext.
119. 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.
120. Leach, P., Salz, R., UUIDS and GUIDS. Internet Draft, available at http://ftp.ics.uci.edu/pub/ietf/webdav/uuid-guid/draft-leach-uuids-guids-01.txt. . 1998.
121. Kermarrec, A., Kuz, I., Van Steen, M., and Tanenbaum, A. A Framework for Consistent, Replicated Web Objects. in Proc. 18th Int'l Conf. on Distributed Computing Systems, IEEE, 1998. pp. 276-284. Cited on page 115. 1998.
122. Gwertzman, J.a.S., M. The Case for Geographical Push-Caching. in Proc. Fifth Workshop Hot Topics in Operating Systems, IEEE, 1996. pp. 51-55. Cited on page 327. 1996.
123. Aktas, M.S., Extended UDDI XML Metadata Service web site, available at http://www.opengrids.org/extendeduddi.
124. XPATH, XPATH XML query language, more information is available at http://www.w3.org/TR/xpath.