Fault Tolerant High Performance Information Services for Dynamic Collections of Grid and Web Services

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

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

Abstract. E-Science Semantic Grids can often be thought of as dynamic collection of semantic subgrids where each subgrid is a collection of modest number of services that assembled for specific tasks. We define a Gaggle as a modest number of managed and actively interacting Grid/Web Services, where services are put together for particular functionality. The information management requirements in *Gaggles* include both the management of large amounts of relatively static services and associated semantic information as well as the management of multiple dynamic regions (sessions or subgrids) where the semantic information is changing frequently. We design a hybrid, fault tolerant, and high performance Information Service supporting both the scalability of large amounts of relatively slowly varying data and a high performance rapidly updated Information Service for dynamic regions. We use the two Web Service standards: Universal Description, Discovery, and Integration (UDDI) and Web Services Context (WS-Context). We evaluate our approach by applying various tests to investigate the performance and sustainability of the centralized version of our implementation that is applied to sensor and collaboration grids. The experimental study on system responsiveness of the proposed approach shows promising results. This study indicates that communication among services can be achieved with efficient centralized metadata strategies, with metadata coming from more than two services. In contrast point-to-point methodologies provide service conversation with metadata only from the two services that exchange information. In addition, our performance indicates that efficient mediator services also allow us to perform collective operations such as queries on subsets of all available metadata in service conversation.

1 Introduction

E-Science Semantic Grids can often be thought of as dynamic collection of semantic subgrids where each subgrid is a collection of modest number of services that assembled for specific tasks such as forecasting earthquakes [1] or managing an audio/video collaboration session [3]. We term an actively interacting (collaborating) set of managed and modest number of services as a *Gaggle* where services are put

together for particular functionality. A particular Semantic Grid may consist of several *Gaggles* each featuring intense local activity with less intense inter-gaggle interactions. Each *Gaggle* maintains the dynamic information which is the session related metadata generated as result of interactions among Grid/Web Services. *Gaggles* are also termed Grid Processes in the China National Grid [27]. They are sessions in the field of collaboration. An infrastructure for the Semantic Grid is discussed in [2] where Grid Processes may be defined as cooperative processes that support the definition, management and integration of business processes. We also note that *Gaggles* may be composed from other "sub" *Gaggles* hierarchically.

Extensive metadata requirements of both the worldwide Grid and smaller sessions or "gaggles of grid services" that support local dynamic action may be investigated in diverse set of application domains such as sensor and collaboration grids. For example, workflow-style Geographical Information Systems (GIS) Grids such as the Pattern Informatics (PI) application [1] require information systems for storing both semi-static, stateless metadata and transitory metadata needed to describe distributed session state information. The PI application is an earthquake simulation and modeling code integrated with streaming data services as well as streaming map imaginary 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, dynamic information systems. GlobalMMCS is a service-oriented collaboration system which integrates various services including videoconferencing, instant messaging and streaming, and is interoperable with multiple videoconferencing technologies. Zhuge defines Knowledge Grid in [30-31] as "an intelligent and sustainable interconnection environment that enables people and machines to effectively capture, publish, share and manage knowledge resources and that provides appropriate on-demand services to support scientific research, technological innovation, cooperative teamwork, problem solving, and decision making". To this end, Gaggles may also be thought of as dynamic sub-components of the Knowledge Grid. Each Gaggle might be created in a dynamic fashion to support science and engineering applications of the Knowledge Grid.

Figure 1 illustrates a model of building system hierarchy where services are aggregated into atomic grids that perform basic functionalities. The basic (atomic) grids include Geographical Information Systems (GIS), collaboration, sensor, compute or 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 figure with services like extended UDDI XML metadata service for static information and WS-Context XML metadata service for dynamic information. The atomic (basic) grids can be re-used in all critical infrastructure grids which in turn customized, compared and overlaid with other grids for different critical infrastructure communities such as crisis grid, emergency response and so forth. As an example, PI grid application can be built in composite fashion from basic grids, such as GIS and sensor grids. Given this picture, we expect that Grid of Grids concept [36] can be applied recur-

sively and dynamically to build grid applications with modest number of services gathered together at any one time to perform particular functionality.

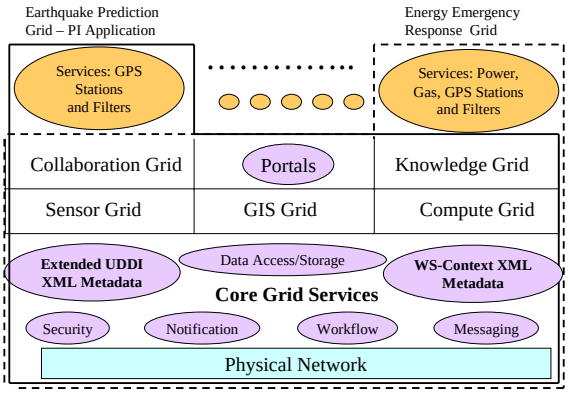


Fig.1. *Gaggles* may be built in a dynamic fashion as Grids of Grids applications with modest number of services involved at any one time for particular functionality

The Grid Information Services support discovery and handling of services through metadata and are vital components of Grids [4]. In this research, we are particularly interested in supporting discovery and handling of metadata for the Gaggles, i.e. one of the sub-grids of the whole Grid, where semantic information is changing frequently. Handling information requirements of these applications requires high performance, fault tolerant information systems. These information systems must be decentralized, relocate metadata to nearby locations of interested entities and provide efficient access, storage of the shared information, as the dynamic metadata needs to be delivered on tight time constraints within a *Gaggle*.

1.1. Motivation

We identify the following problems in Information Services supporting both traditional and Semantic Grids. First, Grid Information Services need to be able to support dynamically assembled service collections gathered at any one time to solve a particular problem at hand such as calculating damages from disruptions at the time of a crisis. Most of the traditional Grid Information Services [5-6] however are not built along this model. Second, Information Services should scale in numbers and geographical area. Most existing solutions [5-6] however have centralized components and do not address scalability and high performance issues. Third, Information Services need to be able to take into account user demand changes when making decisions on metadata access and storage. Fourth, Information Services need to be able to provide uniform interface for publishing and discovery of both dynamically

generated and static information. Existing Grid Information Services however do not provide such capabilities. We therefore see this as an important area of investigation. This paper presents our design of an architecture and prototype to address the identified problems above. We describe a novel architecture for fault tolerant and high performance Information Services in order to manage distributed, dynamic session related metadata while providing consistent, uniform interface to both static and dynamic metadata.

1.2. Requirements

We design our architecture to meet the following requirements:

Uniformity: The types and update frequency of information may vary in both traditional and Semantic Grids. This requires a hybrid Information Service providing a uniform interface to dynamic/static metadata and supporting both the scalability of large amounts of relatively slowly varying information and a high performance rapidly updated Information Service for dynamic regions.

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.

Persistence: Archival of session metadata may provide a metadata management system enabling session failure recovery or replay/playback capabilities for collaboration grids. This requires persistent metadata storage capability.

Dynamism: Dynamic metadata, i.e. rapidly updated and short-lived information need to be supported in both traditional and Semantic Grids. Furthermore, metadata need to be reallocated based on changing user demands and locations. This requires Information Services that can support metadata for dynamic regions and that can provide discovery of data-systems hosting the metadata under consideration in a dynamic fashion.

Performance: The update frequency on short-lived metadata may vary based on applications. Here, the system is required to support dynamic changes with a fine granularity time delay for the systems with a modest number of involved services (say, up to thousand services per session).

1.3. Contributions and Organization

The main contributions of this paper are two-fold. First, we present a novel architecture for a WS-Context [20] complaint metadata catalog service supporting distributed

or centralized paradigms. We use an extended version of UDDI [21] for slowly varying metadata and present a uniform and consistent interface to both short-lived dynamic and slowly varying quasi-static metadata. We explore the application of context (session-related dynamic metadata) management in Grid systems to correlate activities in workflow-style applications, by providing a novel approach for management of widely distributed, shared session-related dynamic metadata. We investigate the problem of distributed session management in Grid applications, by providing an approach for distributed event (session metadata) management system enabling session failure recovery or replay/playback capabilities. We also address lack of search capabilities in Grid Information Services, by providing uniform search interface to both interaction independent and conversation-based metadata enabling service discovery through events.

Our second contribution is the application of topic-based publish/subscribe methods to the problems of dynamic replication methodology to support dynamic metadata. We utilize a multi-publisher, multicast communication middleware and a topic-based publish/subscribe messaging system as a communication middleware to exchange messages between peers.

This paper is organized as follows. Section 2 reviews the state of art in existing information services and replica hosting environments. Section 3 reviews our design for information systems to support *Gaggles* paying particular attention to distributed data management aspects of the system. We discuss the status of the system in section 4 and the evaluation of our prototype in Section 5. In Section 6, we summarize and discuss future work.

2 Background

Peer-to-Peer (P2P) systems may broadly be categorized as pure and hybrid [32]. On one hand pure systems endeavor for total decentralization and self-organization, on the other hand hybrid systems have some form of centralized control such as a look-up service [31]. In this paper, we focus our study on the Information Systems that adopt pure P2P networks which may further be categorized as a) structured and b) unstructured. In structured P2P architectures, system resource placement at peers is enforced with strict constraints which in turn create heavy overhead on the bootstrap of the network. For an example, Globus Monitoring and Discovery System (MDS4) [5] 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) [6] presents a relational model where users query/store/access metadata centrally and if information found directly connect to information providers to retrieve the data without intermediary nodes. Another example is the structured P2P systems where the nodes in the systems are equally enabled and controlled and service information is disseminated to all nodes (CAN [7], Chord [8]). Unstructured P2P architectures can be characterized as systems where there is complete lack of constraints on

the placement of resources and the capabilities of the system nodes. An extensive survey on Grid Information Services can be found at [9, 35].
Architectures with pure decentralized storage models have focused on the concept of distributed hash tables (DHT) [7, 8]. 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. Although the DHT approach provides good performance on routing messages to corresponding nodes, it has various limitations such as primitive query capabilities on the database operations. Here, we design an architecture which can be defined as an unstructured P2P approach to P2P/Grid environment. We use multi-publisher message broadcasting through a topic-based publish/subscribe messaging system, which support access and storage decisions among distributed nodes.

Well-defined descriptions of resources, services and data constitute metadata. Metadata can be represented using varying metadata models such as XML Schema or Semantic Web languages (RDF [28], OWL [29], etc.). Here, we are mainly concerned with managing the metadata and delivering to clients, not with knowledge processing. We presume the metadata models to be application-specific and not defined by us. To this end, we are concentrating on distributed computing problems of managing metadata in the Semantic Grid. See Section 4 of this paper for more discussion.

An approach to solve the problem of locating services of interests is the UDDI Specifications [21] from OASIS (http://www.oasis-open.org). The UDDI is WS-I compatible and offers users a unified and systematic way to find service providers through a centralized registry of services. We identify the following limitations in UDDI Specifications. 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. So, there may be stale data in registry entries. Third, UDDI does not support extensive metadata requirements of rich interacting systems. For instance, services may require an Information Service to publish and discover session metadata generated by one or more services as a result of their interactions. Fourth, since UDDI is domain-independent, it does not provide domain-specific query capabilities such as geo-spatial queries.

There have been some solutions introduced to provide better retrieval mechanism by extending existing UDDI Specifications. UDDI-M [10] and UDDIe [11] projects introduce the idea of associating metadata (name-value pairs) and lifetime with UDDI Registry. UDDI-M^T [12-13] improves this approach in several ways such as improving the metadata representation from attribute name-value pairs into RDF triples to provide semantically rich service descriptions and relevant information. The Grimoires registry project (http://twiki.grimoires.org/bin/view/Grimoires/WebHome) extends the UDDI-M^T to provide a registry which can support multiple service description models by taking into account robustness, efficiency and security issues. Another approach to leverage UDDI Specifications was introduced by METEOR-S [14] project which also utilizes semantic web languages when describing a service

(such as data, functionality, quality of service and executions) in order to provide more expressiveness power and better service match-making process.

In our design, we too extend UDDI information model by providing an extension where we associate metadata with service descriptions. We use (name, value) pairs to describe characteristics of services similar to the UDDI-M and UDDIe projects. We expand on the capabilities that are supported by these projects, by providing domain-specific query capabilities. An example for domain-specific query capability could be XPATH queries on the auxiliary and domain-specific metadata files stored in the UDDI Registry. Another distinguishing aspect of our design is the support for session metadata. Our design supports not only quasi-static, stateless metadata, but also more extensive metadata requirements of interacting systems. UDDI-M^T and METEOR-S projects are example projects that utilize semantic web languages to provide better service matchmaking in retrieval process. This research has been definitely investigated [12-14] and so not covered in our design. We view dynamic and domain-specific metadata requirements of sensor/GIS and collaboration Grids as higher priority.

We use replication, a well-known and commonly used technique to improve the quality of metadata hosting environments, in our architecture. Sivasubramanian et al. [15] give an extensive survey on reviewing research efforts on designing and developing World Wide Web replica hosting environments, as does Robinovich in [16], paying particular attention to dynamic replication. As the nature of our target data is dynamic, we focus on data hosting systems that are handling with dynamic data. These systems can be discussed under following important design issues: a) distribution of client requests among data replicas b) selection of hosting environments for replica placement c) consistency enforcement.

Distribution of client requests is the problem of redirecting a client to the most appropriate replica server. Most existing solutions to this problem are based on DNS-Server such as in [17-18]. These solutions utilize a redirector/proxy server that obtains physical location of 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. Existing solutions, that apply dynamic replication, monitor various properties of the system when making replica placement decisions [18-19]. For instance, Radar [18] replicates/migrates dynamic content based on changing client demands. Spread [19] considers the path between the data-system and client and makes decisions to replicate dynamic content on that path.

The existing solutions to dynamic replication assume all data-hosting servers to be ready and available for replica placement and ignore "dynamism" in the network topology. In reality, data-systems can fail anytime and may present volatile behavior.

We use a pure Peer-to-Peer approach, which is based on multi-publisher multicast mechanism, when distributing access and storage requests to data-systems.

The **consistency enforcement issue** has to do with ensuring all replicas of the same data to be the same. Various techniques have been introduced in consistency management. For instance, the Akamai project [17] introduces versioning where a version number is encoded to document identifier, so that client would only fetch the updated data from the corresponding data hosting system. Radar [18] applies primary-copy approach where an update can be done only on the primary-copy of the data.

In our design, we employ a strategy which suggests propagation of updates only if it is necessary. Our main approach is to provide client-centric consistency which provides guarantees for a single client's access to a replicated data store. We use Network Time Protocol (NTP) clients to achieve synchronized timestamps to give labels, i.e. versions, to each context stored in the system.

Our architecture differs from web replica hosting systems as the intended use of our architecture is not to be a web-scale hosting environment. The scale of our target systems is in the order of a few dozen to at most a thousand entities participating in a session. Our target domains range from collaboration systems such as GlobalMMCS project to geographical information systems such as Pattern Informatics GIS-Grid. The participant entities of these systems might dynamically generate metadata during a session. Such metadata can be expected to be small in size and big in the volume depending on the Grid application.

3 Information Services

We have designed a novel architecture to Information Services presenting a uniform interface to support handling and discovery of not only quasi-static, stateless metadata, but also session related metadata. Our approach is to utilize the existing state-of-art systems for handling and discovering static metadata and address the problems of distributed management of dynamic metadata. In order to be compatible with existing Grid/Web Service standards, we based the interface of our system on the WS-Context [20] and UDDI [21] Specifications. We have extended and integrated both specifications to provide uniform and consistent service interface to both dynamic and static metadata. A centralized version of our architecture is depicted in Figure 2. In our design, we use replication technique to provide fault tolerance, load balancing, reduced access latency and bandwidth consumption. In order to enable communication between replica hosting servers, we utilize a topic based publish-subscribe mechanism to provide message-based communication as depicted in Figure 3. Figure 3 illustrates two clients interacting with an hybrid Information Service that provides a uniform programming interface to both quasi-static and dynamic metadata. In this scenario, on receiving the client requests, the system first extracts dynamic and static portions of the query. The static portion of the query is simply forwarded to UDDI XML metadata service, while the dynamic part is handled by the Information

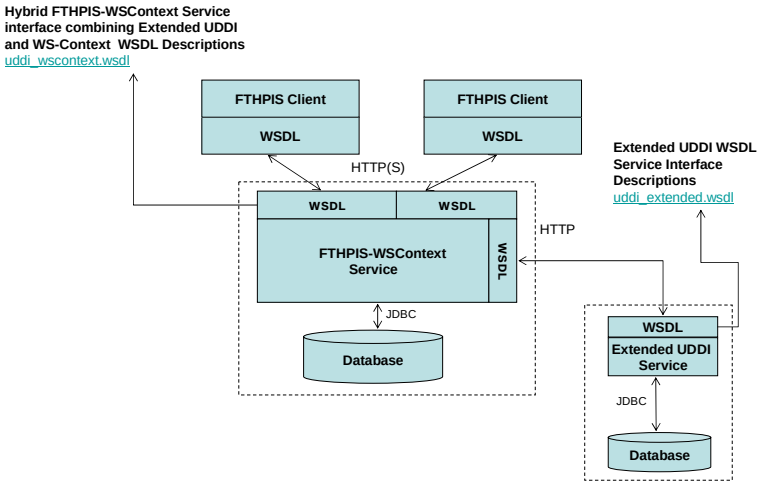


Fig.2. Our design integrates both UDDI and WS-Context Web Service Interfaces to provide a uniform programming interface to service metadata. This figure illustrates the centralized version of FHTPIS-WSContext Service interacting with two clients. The service also interacts with an external extended UDDI service when the incoming inquiry requests require static metadata.

Service itself. If the query asks for external metadata, then the query is multicast to available replicas through a topic-based publish-subscribe mechanism. On receiving the responses from both dynamic and static metadata spaces, the system returns the results to querying clients. In the following sub-sections, we discuss different aspects of our architecture as following. First, we discuss the details of how we have extended and combined UDDI and WS-Context specifications. Then, we describe the fault-tolerant aspects of our design in details. Next, we discuss the software multicast communication mechanism followed by the architectural components of the proposed system.

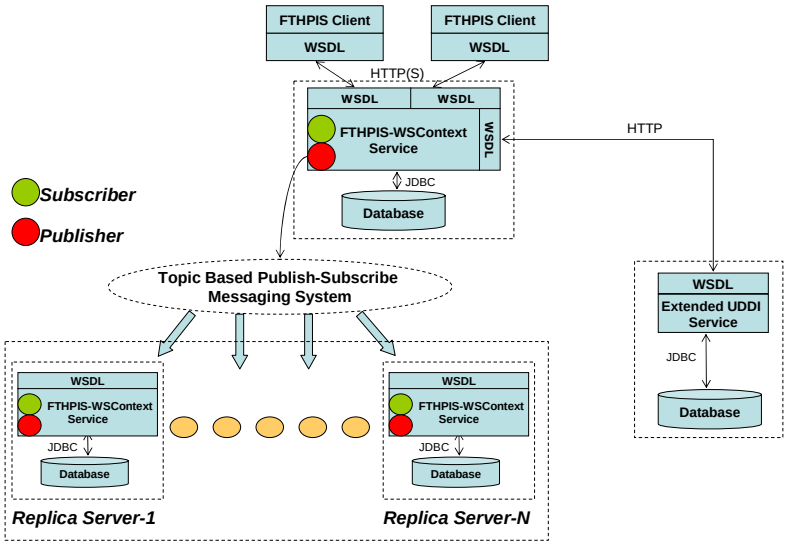


Fig.3. This figure illustrates decentralized Fault Tolerant High Performance Information Services (FTHPIS) from the perspective of a single FTHPIS-WSContext Service interacting with two clients. The FTHPIS system uses a topic based publish subscribe messaging system to enable communication between the services.

3.1. Extended UDDI XML Metadata Service

We have extended existing UDDI Specifications to annotate service descriptions with metadata describing characteristics of services.

We designed a data model for service metadata by extending UDDI Data Structure Schema. Detailed design documents can be found at http://www.opengrids.org/-extendeduddi/index.html. Based on this model each service entry is associated with an XML tag called “metadataBag” which consists of one or more “serviceAttribute” sub-elements. Here, each “serviceAttribute” corresponds to a piece of metadata and it is simply expressed with (name, value) pairs. The value type of a “serviceAttribute” can either be string or integer. As an example, we can illustrate a “serviceAttribute” as in ((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 service attribute may correspond to a domain-specific metadata where service metadata could be directly related with functionality of the service. For instance; OGC compatible GIS services provide a “capabilities.xml” metadata file describing

the data coverage of geo-spatial services. We use an “abstractAttributeData” element to represent such metadata and store/maintain these domain specific auxiliary files as-is. Here, the abstract attribute data could simply be in any representation format such as XML or RDF.

In order to support/integrate quasi-static stateless metadata in UDDI Registries, we also extended existing UDDI XML API Schema and implemented it as a web service. We introduced metadata-oriented publishing/discovery capabilities by expanding on existing UDDI API such as “save_service”, “find_service” and “get_serviceDetail”. We also introduced additional API to let third party users of services a) attach additional metadata and b) pose queries for particular metadata to already published service entries. These additional API are “get_serviceAttributeDetail”, “save_serviceAttribute”, “find_service-Attribute” and “delete_serviceAttribute”. Further design documentation on XML APIs is available at http://www.opengrids.org/-extendeduddi/index.html.

Our design may also support discovery of domain-specific prescriptive metadata as in the following scenario. A querying user constructs a query “metadataBag” consisting of a list of “serviceAttribute”s. Each “serviceAttribute” forms a search criterion. The constructed “metadataBag” is passed to UDDI Registry as an argument of the extended “find_service” function. We implement “find_service” functionality in a way to support XPATH query capabilities on the UDDI Registry. Say, in given a “serviceAttribute” element, one could indicate a) XPATH query statement and b) name of the prescriptive metadata file. If the search criterion is a XPATH query, then the query is applied on the corresponding auxiliary file stored in the UDDI Registry. The results will be a list of services that satisfy user’s query. This way, we can apply domain-specific queries such as geo-spatial queries on the metadata services.

Given all these capabilities, one can simply populate the registry with metadata as in following scenario. Say, a user publishes a new service into UDDI Registry. In this case, the user constructs a “metadataBag” filled with “serviceAttributes” where each “serviceAttribute” has (name, value) pairs. Each pair may describe one generic descriptive characteristics of the service such as throughput, or usage cost. If a service metadata is domain-specific, we use an “abstractAttributeData” element and express the serviceAttribute as a (name, abstractAttributeData) pair. As the “metadataBag” is constructed, it can be attached to the new service entry which can then be published with extended “save_service” functionality that we introduced.

As we research UDDI Specifications to integrate with our system, we have encountered various limitations in its capabilities which we address in a separate paper [22]. Our work on UDDI is for a specific type of metadata: semi-static and context-free. UDDI is appropriate for data that is long-lived (i.e. should be true for months or years) and that is independent of the client interaction (i.e. all clients issuing the same requests get the same responses). We discuss the parts of our architecture that supports dynamic information in the short-lived service collections in the following section.

3.2. Extended WS-Context XML Metadata Service

We have extended the WS-Context Specification [20] to manage session metadata between multiple participants in Web Service interactions.

We designed a data model for managing dynamic metadata by extending existing WS-Context XML Schema. Detailed design documents can be found at http://www.opengrids.org/wscontext/index.html. Based on this model, we define session entity which may be considered an information holder; in other words, a directory where context with similar properties are stored. Each session entry is associated with an XML tag called "contextBag" which consists of one or more "Context" sub-elements. Here, a context entity is used to represent dynamic metadata and "contextBag is considered as metadata collection associated with a session. Each context 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 A. The value of a context is stored as SQL BLOB type in the MySQL database. A "context" 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. Each session entity has a "session-directory-metadata". Session directory metadata describes the child and parent nodes of a session. This enables the system to track the associations between sessions.

In order to support/integrate dynamic metadata, we also extended existing WS-Context XML API and implement it as a web service. We introduced various additional publishing/discovery capabilities to enable the system to track the associations between sessions and contexts by expanding on primary functionalities of WS-Context XML API such as "setContext" and "getContext". Here each context is stored and retrieved associated with a session. The additional XML API is designed to let third party users a) to locate/retrieve/save/delete contexts associated to a session and b) to locate/retrieve/save/delete particular sessions with given contexts and/or participating session-entities. Extended version of WS-Context XML API include "find_context", "get_contextDetail", "save_context", "delete_context", "find_session", "get_sessionDetail", "save_session", and "delete_session". Further design documentation on WS-Context XML API is available at http://www.opengrids.org/-wscontext/index.html.

3.3. An hybrid Information Service interface combining both extended UDDI and WS-Context functionalities

We combine both extended UDDI and WS-Context implementations within a hybrid service. Our aim is to provide a uniform service interface to service metadata. To this end, we introduced hybrid publishing/discovery capabilities, such as “save_service”, “find_service”, “delete_service” and “get_serviceDetail”, supporting both dynamic and quasi-static, stateless service metadata.

Given these capabilities, one can simply populate this hybrid information service with service metadata as in the following scenario. Say, a user publishes a new service into the system. In this case, the user constructs both “metadataBag” filled with “serviceAttributes” and “contextBag” filled with “contexts” where each context describes the sessions that this service will be participating. As both the “metadataBag” and “contextBag” is constructed, they can be attached to a new “service” element which can then be published with extended “save_service” functionality of the hybrid Information Service. On receiving publishing service metadata request, the system applies following steps to process service metadata. First, the system separates the dynamic and static portions of the metadata. Then, the system delegates the task of handling discovery of static portion (“metadataBag”) to extended UDDI service. Next, the system itself provides handling and discovery using dynamic portions of the metadata in the metadata replica hosting environment. Further design documentation on hybrid Information Service XML API is available at http://www.opengrids.org/-extendeduddi/index.html.

The intended use of our approach is to support information in dynamically assembled Semantic Grids where “real-time” decisions are being made on which services to tie together in a dynamic workflow to solve a particular problem. One may think of WS-Context complaint Information Services as the metadata catalog for semantic metadata as in an RDF triple store. The semantic metadata expresses the relationships between resources, while the applications that access the metadata catalog deduct further (inferred) information. In our design, the distinctive semantic richness comes from the highly dynamic architecture with metadata from more than two services (in contrast WS-Transfer, WS-Metadata Exchange Specifications that only easily get semantic enhancement from the two services that exchange metadata). We discuss various research issues in building Information Services for dynamically assembled Semantic Grids in the following section.

3.4. Fault Tolerant High Performance Information Services

We have considered two application domains from collaboration and sensor/GIS grids to demonstrate the use of our system: GlobalMMCS and PI GIS-Grid. GlobalMMCS is a peer to peer collaboration environment where videoconferencing sessions can take place. Any number of widely distributed services can attend to a collaboration session. GlobalMMCS requires persistent archival of session metadata

to provide replay/playback and session failure recovery capabilities. The PI GIS-Grid is a workflow-style Grid application which requires storage of transitory metadata needed to correlate activities of participant entities. Both application domains require a decentralized metadata hosting environment which can support both scalability (of large amounts of information) and performance requirements (of rapidly updated dynamic information). To this end, we identify two important research issues that need to be answered in our design: fault tolerance and high performance.

We use replication technique to provide fault tolerance, which improves the quality of our data hosting environment. If one of the redundant storage elements goes down, it automatically consults remaining elements to restore itself. The replication technique can also lead into high performance by reducing a) bandwidth consumption and b) the time between a client issuing a request and receiving the corresponding response.

One approach to replication is replicating context in every node in the distributed system architecture. This full-replication method could surely provide best fault-tolerance in terms of availability. However, this approach doesn't scale. The more replicas need to be kept consistent, the higher quantity of exchanged messages and time required. The other approach is partial-replication which suggests replication of contexts only if it is necessary to minimize the cost needed to keep replicas consistent. So, we choose partial-replication over full-replication.

Replication can also be categorized by the manner in which the replicas are created and managed. On one hand, static replication suggests a strategy where replicas are to be manually created and managed. In a dynamically assembled Gaggle environment, it is not feasible to manually replicate dynamically generated metadata. On the other hand, dynamic replication suggests replication of contexts based on changing user behavior. To this end, as the nature of our data is very dynamic, we use dynamic data replication technique, where data replicas may be created, deleted, or migrated among hosting data-systems based on changing user demands [16].

An example of 11-node based FTHPIS replica hosting environment is depicted in Figure 4 where dynamic metadata (contexts ranging from A to O) replicated on the FTHPIS nodes ranging from 1 to 11. Our main interest in dynamic replication is to place context replicas in the proximity of requesting clients by taking into account changing demand patterns to minimize the response latency. The number and the placement of replicas may change due to demand changes. In the example, the quantity of 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 context space, but the individual contexts based on their demands. Next, we discuss the two important aspects of dynamic replication are access and storage algorithms.

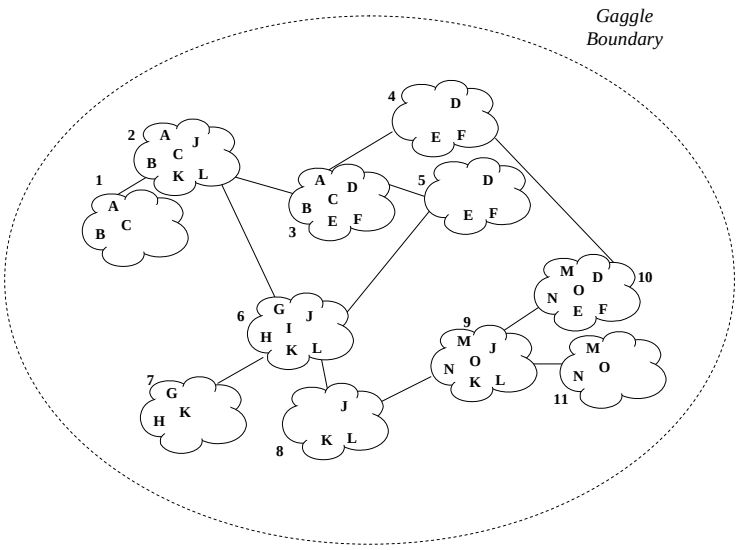


Fig.4. An eleven-node based FTHPIS replica hosting environment. Numbered callout shapes represent replica servers. Letters ranging from A to O correspond to contexts replicated on the replica servers ranging from 1 to 11.

3.5. Access Algorithm

The access algorithm distributes client requests to appropriate replica hosting data-systems. Our model is based on pure Peer-to-Peer approach where each node can probe all other nodes in the network to look up metadata. A primary role of the access algorithm is the discovery of one or more data-systems hosting the requested meta-data. This discovery process consists of two steps: data-system discovery and access. The first step concerns with selection of data-systems that can answer the client requests. The second step is to inform the data-system that is most appropriate for handling the request. In the first step, to find metadata, a node sends a probe message to all other nodes through a software multicast mechanism; target data-systems that host the metadata matching the probe send a response directly to requestor node. Here, response message consists of information regarding how well the data-system can handle this query. For instance, such information may include proximity information between the client and the data-system. On receiving response messages, the requestor node chooses the most appropriate data-system that can handle the request. In the second step, the requestor node sends the client request to the chosen data-system particularly asking to handle the request.

3.6. Storage Algorithm

Storage algorithm selects data-systems for replica placement and decides how many replicas to have in the system. In our design, storage decisions are made autonomously at each node without any knowledge of other replicas of the same metadata. The storage decision is made based on the client requests served by that node. Storage process consists of two separate steps such as metadata placement and metadata creation. The first step has to do with selection of data-systems that should hold the replica and the second step has to do with metadata replica creation. In the first step, each node (data-system) runs the storage algorithm which defines client request thresholds for replica creation and deletion. If a metadata entry is in high demand which is above a pre-defined threshold, then the metadata is replicated. If a metadata entry is in low demand which is below a pre-defined threshold, it will be deleted. To replicate metadata, a node sends a "storage" message to all other nodes through a software multicast mechanism; target data-systems, that have available space, send a respond to directly requestor node. Here, the response message consists of various decision metrics such as client proximity information. On receiving the response messages, replica placement algorithm chooses the most appropriate data-system to replicate the metadata. In the second step, the requestor node sends a replica creation message directly to the chosen data-system asking to store a replica of metadata in consideration. This process creates a dynamic metadata storage in which metadata is moved based on changing client demands.

3.7. Multi-publisher Multicasting Communication Middleware

An importing aspect of our system is that we utilize software multicasting capability which is an important communication medium supporting the ability to send out access and storage requests to the nodes of the system. Any node can publish and subscribe to topics which in turn create a multi-publisher multicast broker network as communication middleware. Here, the publisher does not need to know the location and identities of receivers. It publishes a message to a topic to which all nodes subscribe.

The architectural design of the proposed system is built on top such publish/ subscribe based multicast broker network system as depicted in Figure 5. In this illustration, each peer runs a FTHPIS-WSContext Information Service whose detailed architecture is also given in Figure 5. We use NaradaBrokering (NB) [23] 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.

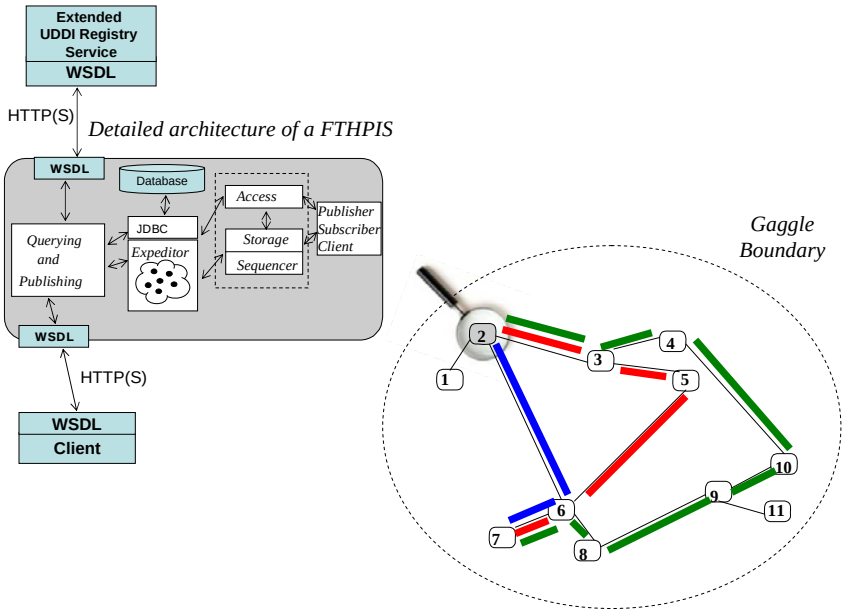


Fig.5. An example eleven-node based FTHPIS replica hosting environment where each node is connected with publish-subscribe based overlay network. Numbered squares represent nodes running FTHPIS-WSContext Service (see figure 2 for centralized version of the service) whose detailed architecture is also illustrated in the figure. The tick lines on the figure are used to show different message delivery routes between peers 2 and 7.

The organization scheme of this scenario is small world network [37, 38] where the communication between peers increases logarithmically with geometric increase in network size, as opposed to exponential increase in uncontrolled settings [23]. We particularly use NaradaBrokering software in our design, since it provides efficient message delivery to the targeted peer en route to intended clients. For example, in figure 5, we observe various message delivery routes from peer-2 to peer-7. The NaradaBrokering software is able to make decision to choose 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.

3.8. System Components

Our proposed architecture consists of various modules such as Query and Publishing, Expeditor, Access, Storage and Sequencer Modules. Architectural design of our system is illustrated in the upper-left corner of the Figure 5.

3.8.1. Context Query and Publishing Modules: These modules receive client requests through a uniform service interface for publishing/discovering dynamic and static metadata. The client query/publishing requests are processed and dynamic metadata parts of the queries are extracted. Then, the request is forwarded to Expeditor Module to find the results. Likewise, static metadata portion of the requests is relayed to external UDDI Service to publish/discover services through static metadata.

3.8.2. Expediter Module: This is a generalized caching mechanism. Each node has a particular expediter. One consults the expediter to find how to get (or set) information about a dataset in an optimal fashion. The expediter is roughly equivalent to replica catalog in classic Grids. Expeditor forms a built-in memory and it maintains Context metadata objects in Context Spaces. A Context Space is an implementation of Tuple-Spaces concept [24]. Context Spaces allow us to apply space based programming to provide mutual exclusive access, associative lookup and persistence.

3.8.3. Access Module: This module runs the access algorithm mentioned above. It support request distribution by publishing messages to topics in NB network. It also receives messages (in respond to client request) coming from other peers and forward these query messages to Expediter Module. The Access Module locates the nodes that are closest in terms of network distance with lowest load balance from the node requesting access to the communal node in question. It also takes into account the load balance of each responding data-system when choosing the right data-system.

3.8.4. Storage Module: This module runs the storage algorithm. It interacts with the Expediter Module and applies the storage algorithm to all local Context metadata. If the metadata is decided to be replicated, then the storage module advertises this replication by multicasting it to available peers through NB publish/subscribe mechanism. The storage module also interacts with the Sequencer module in order to label each incoming metadata with a time stamp.

3.8.5. Sequencer Module: This module ensures that an order is imposed on actions/events that take place in a session. The Sequencer Module interacts with the Storage Module and labels each metadata which will be replicated in this replicated metadata hosting environment. The Sequencer Module interacts with Network Time Protocol (NTP) clients to achieve synchronized timestamps among the distributed nodes.

When receiving a query, the Query Module first processes the query and extracts the dynamic metadata portion of the query. Then, the Query Module forwards the query

to Expediter, where the Expeditor Module checks whether the requested data is in Context Spaces. If the Expeditor Module can not find the result in Context Space or if the requested metadata is expired, then the query is forwarded to the JDBC Handler to query the data in local database. If the query asks for external metadata, then the Expediter will forward the query to Access Module, where the Access Module multicast a probe message to available Information Services through NB and communicates with the Information Services that are the original data sources for this query. The query is responded by an Information Service which may be the best qualified Information Service is to handle this query.

4 System Status

Extended UDDI XML Metadata Services: We have implemented extended UDDI XML Metadata Services [25] handling and discovery of static metadata based on the WS-I standard Uniform Description, Discovery, and Integration (UDDI) Specifications. We base our implementation on jUDDI (version 0.9r3), a free, open source, and java implementation of the specification. (More at http://www.juddi.org). jUDDI has been architected to act as the UDDI front-end on top of existing databases.

In our design, we only use a portion of the jUDDI library as UDDI-front end in order to implement extended version of UDDI XML API. We have discarded jUDDI servlet-based architecture and implemented Grid/Web Services interfaces as front access to UDDI Registries. We have enhanced jUDDI in the following ways. First, we expanded on UDDI XML Data Structure and implemented extensions to UDDI XML API to associate metadata with service entries. Second, we implemented a leasing capability. This solves a problem with UDDI repositories: information can become outdated, so we automatically clean up entries by assigning them an expiration date. Leases on metadata may be extended. Third, we implemented GIS-specific taxonomies to describe Open GIS Consortium (OGC) compatible services such as Web Feature Services and their capabilities files. The “capabilities.xml” file is (in effect) the standard metadata description of OGC services. Finally, we implemented a more general purpose extension 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 (http://www.w3.org/TR/xpath). This allows us to support other XML-based metadata descriptions developed for other classes of services besides GIS. 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.

WS-Context complaint XML Metadata Services: We have implemented a centralized version of WS-Context complaint XML Metadata Services [25] handling discovery of dynamic, session related metadata. Here, session related metadata is short-lived and dependent on the client [26]. The WS-Context metadata service keeps track of context information shared between multiple participants in Web Service interactions. The context here has information such as unique ID and shared data. It allows a col-

lection of action to take place for a common outcome. We utilize WS-Context Specification to maintain user profiles and preferences, application specific metadata, information regarding sessions and their participating entities. Each session is started by the coordinator of an activity. The coordinator service publishes the session metadata to Information Service and gets a unique identifier in return. The uniqueness of the session-id is ensured by the Information Service. Sessions can obviously be composed from other "sub" sessions hierarchically. Here, each session is associated with the participant services of that session. Dynamic session information, i.e. context, travels within the SOAP header blocks among the participant entities within the same activity. Our implementations of UDDI and WS-Context Metadata Services do not use XML databases but for efficiency convert the XML to SQL and store in MySQL database.

Hybrid Information Service Interface combining extended UDDI and WS-Context functionalities: We assume a range of applications which may be interested in integrated results from two different metadata spaces; UDDI and WS-Context. When combining the functionalities of these two technologies in one hybrid service, we may enable uniform query capabilities on context (service metadata) catalog. To this end, we have implemented a uniform programming interface, i.e. a hybrid information service combining both extended UDDI and WS-Context. On receiving service-metadata publishing/inquiry requests, the hybrid service simply delegates the task of handling metadata to appropriate end.

5 System Evaluation

We designed various experiments to investigate the performance of the centralized version of the FTHPIS-WS-Context Information Service. In the system evaluation section, we are particularly addressing following research questions:

- What is the baseline performance of the FTHPIS implementation for given standard operations?
- What is the effect of the network latency on the baseline performance of the system?
- What is the performance degradation of the system for standard operations when processing more users/transactions simultaneously at various loads?
- What is the performance of the system in a testing environment that is designed based on targeted application use domains where there are both multiple clients and providers running simultaneously at various loads?
- What is the effect of Operating System CPU thread scheduling interference on the system performance?

To evaluate the performance of the system, we used response time as the performance metric. The response time is the average time from the point a client sends off a query till the point the client receives a complete response. Although, there is much func-

tionality introduced by the FTHPIS- WS-Context Service system, we focus our experiments on the publication and inquiry capabilities. We test the performance of our implementation with respect to response time at both the querying client and publishing provider applications. In the following section, we give details of the environment of our experiments.

5.1. 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. In addition to these nodes, we also used a desktop machine (kilimanjaro.ucs.indiana.edu) where we ran our client application. The cluster computers were equipped with Intel® Xeon™ CPU (2.40GHz) and 2 GB RAM. Each of the machines ran Linux kernel 2.4.22. The desktop machine ran Windows XP and was equipped with Intel Pentium 4 CPU (3.4 GHz) and 1 GB RAM. The network bandwidth between these machines was 900 Mbits/sec.[1]

We tested the performance of the FTHPIS with a client program called WSContextClient (a program for sending queries to WS-Context Service) and a provider program WSContextProvider (a program for publishing context to FTHPIS). Both WSContextClient and WSContextProvider are multithreaded programs. These applications take following arguments: a) the number of threads, b) the number of queries/publications to be executed, and c) the time to wait after each transaction. When creating multiple threads we use a barrier to stop the threads until a specific number of threads is at hold. Then the threads are released and can continue. This allowed us to simulate concurrent querying/publishing accesses to the server. We illustrate timing methodology in the pseudo code below.

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

```
SET the number of threads to N
SET the number of transaction to be executed to T
SET the time to wait after each transaction to S

CREATE N number of threats
STOP the threads until N threads is created and ready

FOR X = 1 to T
    SET start to 0, stop to 0
    SET start to getTimeMicroseconds()
    saveContext(…) or getContext(…)
    SET stop to getTimeMicroseconds()
    PRINT  (stop – start)
     IF S is set THEN
        WAIT for S time interval before next transaction
     END IF
 END FOR
```

In the experiments, the FTHPIS was running on cluster node-6, while the WSContextClient was running on kilimanjaro.ucs.indiana.edu. We ran the WSContextProvider applications across the cluster nodes 1 to 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 the experiments, we used metadata samples (which were actually used in aforementioned Pattern-Informatics application use domain) with a fixed size of 1.2KB. We illustrate the WS-Context and UDDI XML metadata samples in appendix A and B respectively. In this work, we assumed XML metadata as flat contexts, i.e. no parent-child relationships existed between contexts stored in the system. We wrote all our code in Java, using the Java 2 Standard Edition compiler with version 1.4.2. In the experiments, we used Tomcat Apache Server with version 5.5.8 and Axis software with version 1.2beta3 as a container for deployment of WS-Context Service. The Tomcat Apache Server uses multiple threads to handle concurrent requests. In the experiments, we increased the default value for maximum number of threads from 150 to 500 to be able to test the system behavior for high number of concurrent clients. We choose to use getTimeMicroseconds() function which is provided by NaradaBrokering [23] software because of its high resolution. As backend storage, we use MySQL database with version 4.1.

5.2. Experiment 1- Responsiveness Experiment

Our primary interest in doing this experiment is to understand the baseline performance of the implementation of FTHPIS- WS-Context Service. We also investigated the effect of network latency on the system performance. We evaluated the perform-

ance of the service for inquiry and publication functions under normal conditions, i.e., when there is no additional traffic.

In this experiment, we investigated four different testing cases: a) a single client sends queries to a FTHPIS node where there is cache hit in the Expeditor module (which is explained in sub-section 3.8.2), b) a single client sends queries to a FTHPIS node where there is cache miss and the query is responded with database access, c) a client sends queries to a UDDI, and d) a client sends queries to a dummy service where the round trip message is extracted to and from container but no processing is applied. The dummy service receives the query/publication request message that is used in previous testing cases and then sends it back to the client without processing it. This test is done to measure the pure network latency of a given operation. At each testing case the client sends 100 sequential queries and average response time was recorded. We repeated same testing cases for publication function as well. In this experiment, we investigated the system performance compared with UDDI registry for given standard operations. In evaluating the UDDI performance, we used the jUDDI registry with following exception. We have discarded jUDDI servlet-based architecture and implemented a Web Service WSDL interface as front-end access to investigate UDDI in similar set up to WS-Context for basic performance. We assume that the underlying structure of jUDDI is similar to any other UDDI registry implementation as they all are implementing the same specification. We tested UDDI inquiry/publication functionalities with an XML metadata size of 1.2 KB which is the same message size used in WS-Context testing cases. The designs of these experiments are depicted in Figures 6-7. Figures 8 and 9 illustrate the system performance when the WS-Context inquiry function was executed, while Figures 11 and 12 illustrate the same when the WS-Context publication function was executed 120 times sequentially. The detailed statistics corresponding to these tests are listed in Table 1 and Table 2 respectively. When we investigated the resulting round trip times, we observed two working modes: startup and initialized. We note that both inquiry and publication functions require more cost at the startup mode. To this end, we only took into account the initialized mode in calculating the average response time for each function. Here, we measured the average response time by considering last hundred observation time samples. We repeated these tests in five different test sets.

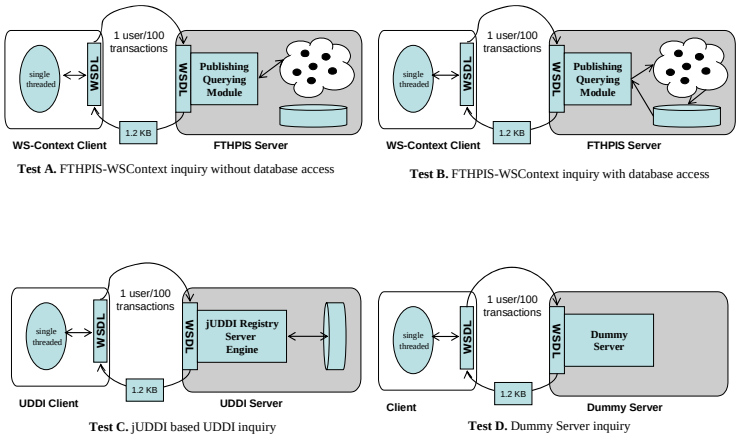
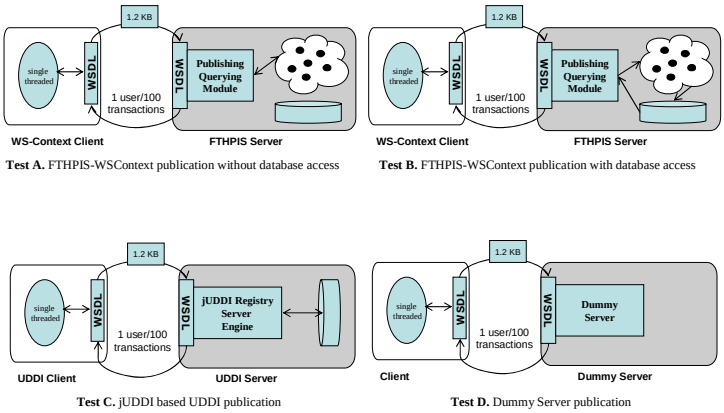


Fig.6. Testing cases of responsiveness experiment for **inquiry functionality**

Results of the Responsiveness Experiment: Figures 10 shows the performance results of inquiry function, while Figure 13 shows the performance results of publication function. It was anticipated that the memory built-in caching mechanism (the Expeditor Module) would improve the performance of the inquiry and publication functions. The empirical results shows that a) for inquiry function, we gain around 18% performance increase (in average) and b) for publication function, we gain around 22% performance increase (in average) by employing a cache mechanism in our design. We observe that UDDI inquiry function executed at an average of 39 milliseconds. A similar performance baseline test has been applied on jUDDI registry for inquiry function in [33] in which the average responsiveness of the inquiry function was measured around 40 milliseconds which in turn helps validating our findings. Based on the experiments performed, we note that cache-enabled WS-Context inquiry function performed with 21% performance increase compared to UDDI-inquiry function. Likewise, cache-enabled WS-Context publication function performed with 30% performance increase compared to UDDI-publication function.

By comparing the results of test case a) for inquiry and publication functions, we observe that publication function requires more time compared to inquiry function. (As mentioned earlier in sub-section 3.8.2, we implemented the cache based on Tuple-spaces paradigm [24].) This is because while the “read” operation can return the value of the context while the context is in the Expeditor module, an update to Expeditor module entry requires a process to look up for context, physically remove the entry, modify its value, and place the copy back into the Expeditor. This in turn increases the time for publication. Likewise, by comparing the results of test case b) for inquiry and publication functions, we observe that publication requires more time

because of the database commit that must take place. By comparing test case d) and test case a) of inquiry function, we note that the network latency costs more than actual time needed to complete the operation in the server-end. One should keep in mind that the performance measurement taken on a tight cluster. However, in a wide area network, one could expect the single server solution to be a bottle-neck for system performance. So, we determined that that network latency may have a significant impact on the centralized version of FTHPIS-WSContext system performance. We conclude that having a built-in caching mechanism provides significant performance increase for given standard operations.



Test A. FTHPIS-WSContext publication without database access

Test B. FTHPIS-WSContext publication with database access

Test C. jUDDI based UDDI publication

Test D. Dummy Server publication

Fig.7. Testing cases of responsiveness experiment for **publication functionality**

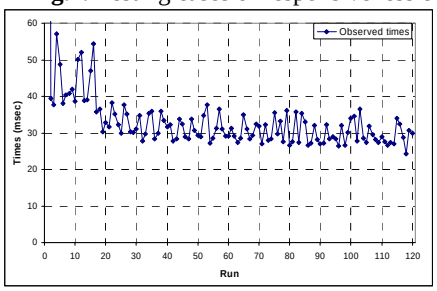


Fig.8. FTHPIS-WSContext inquiry without database access

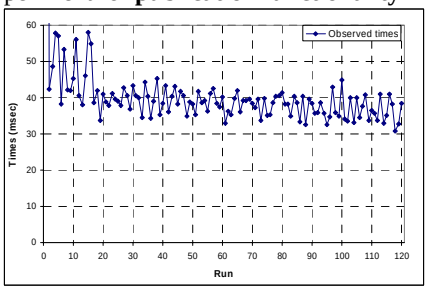


Fig.9. FTHPIS-WSContext inquiry with database access

Statistics for initialized mode of the inquiry performance

Statistics (milliseconds)	Figure 8	Figure 9
Maximum	38.185	45.241
*** Average**	**30.659**	**38.073**
Minimum	24.348	30.824
Standard Deviation	3.191	9.315

Table.1. Statistics for initialized mode of inquiry requests

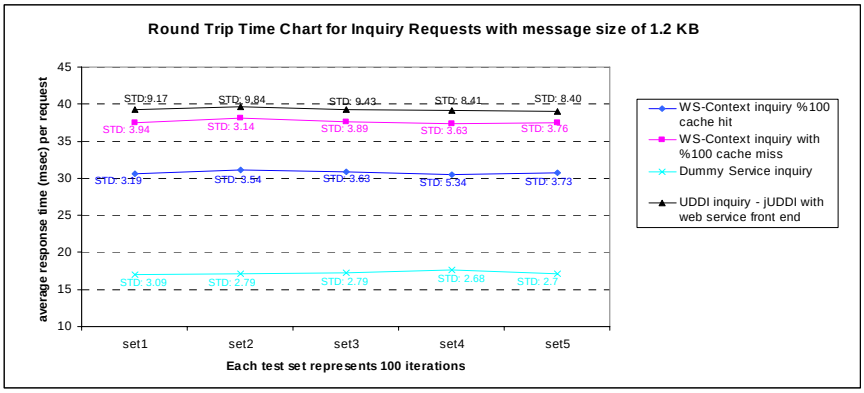


Fig.10. Round Trip Time Chart for Inquiry Requests

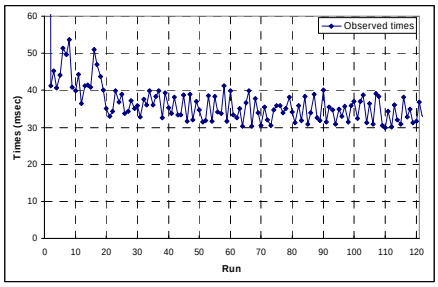


Fig.11. FTHPIS-WSContext publication without database access

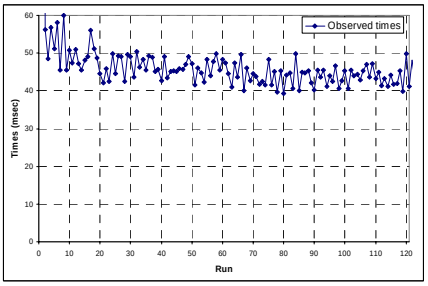


Fig.12. FTHPIS-WSContext publication with database access

Statistics for initialized mode of the publication performance		
Statistics (milliseconds)	Figure 11	Figure 12
Maximum	41.204	56.51
*** Average**	**34.789**	**45.304**
Minimum	29.968	38.925

Standard Deviation	3.009	3.613

Table.2. Statistics for initialized mode of publication requests

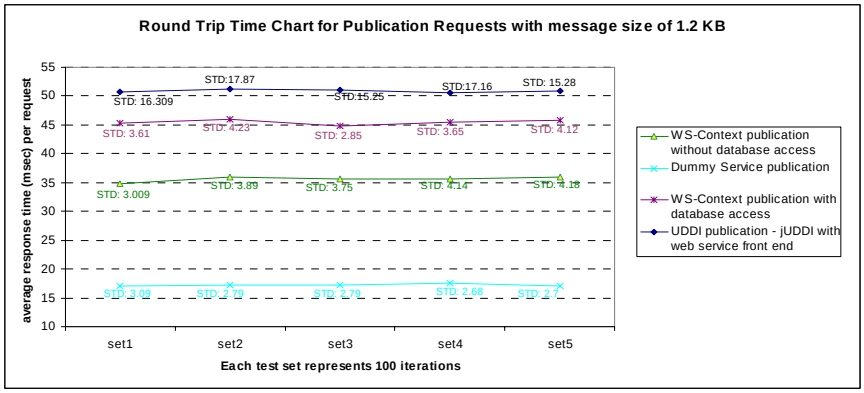
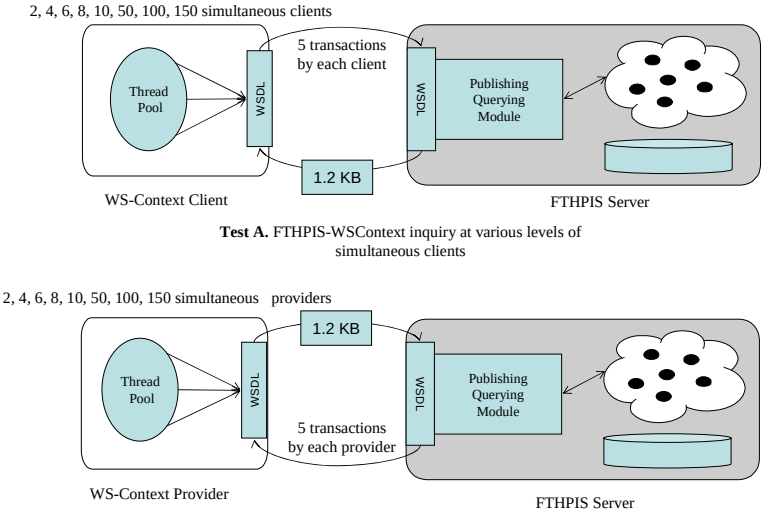


Fig.13. Round Trip Time Chart for Publication Requests

5.3. Experiment 2 - Overloading Experiment:

In the second experiment, 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 FTHPIS-WSContext Service functionalities with additional concurrent traffic. One should keep in mind that we want to test sustainability of the system under the worst case which is the testing of the system by sending queries at the same time from concurrent users with a constant load.

We have done this by gradually ramping-up the number of querying WSContextClients until the system response time degrades. In this experiment, the inquiry function was executed with constant frequency (5 sequential inquires per second) by each client and average service time is recorded at various levels of simultaneous clients. We applied the same testing methodology for publication function to investigate system performance against simultaneous publication requests. The design of this experiment is depicted in Figure 14, while the results are depicted in Figures 15-16.



Test A. FTHPIS-WSContext inquiry at various levels of simultaneous clients

Test B. FTHPIS-WSContext publication at various levels of simultaneous providers

Fig.14. Testing cases of overloading experiment

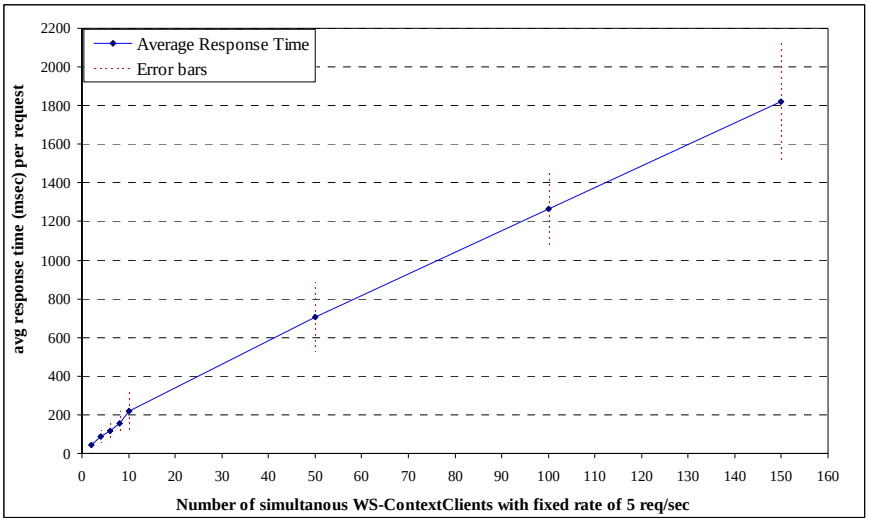


Fig.15. Average WS-Context Inquiry Response Time Chart - making constant load of 5 requests/second by each client

# of simultaneous users	2	4	6	8
fail status	no fail	no fail	no fail	no fail
Statistics (milliseconds)				
Maximum	59.196	140.099	183.847	288.999
***Average**	**46.039**	**88.547**	**115.309**	**157.766**
Minimum	31.268	41.477	47.711	48.999
Standard Deviation	9.894	35.048	40.344	59.699

# of simultaneous users	10	50	100	150
fail status	no fail	no fail	no fail	no fail
Statistics (milliseconds)				
Maximum	465.483	1647.908	1788.579	3100.219
***Average**	**221.417**	**704.325**	**1263.355**	**1821.024**
Minimum	49.042	265.762	821.708	1070.225
Standard Deviation	116.029	179.784	187.987	306.738

Table.3. Statistics of the experiment depicted in Figure 15. These measurements were taken with FTHPIS-WSContext Service without database access. Performance of single client is given in Table 1.

Results of the Overloading Experiment: Based on the results depicted in Figure 15 and Table 3, we determine that the large number of concurrent inquiry requests may well be responded without any error by the system and do not cause significant overhead on the performance. However, we observe that after 100 concurrent users, response time degradation becomes noticeable. We applied the same testing methodology to publication function under two conditions: a) concurrent publishers send their requests when request is handled in Expeditor module (explained in sub-section 3.8.2) without database access, and b) concurrent publishers send their requests when request was handled with database access. Having too many concurrent queries on MySQL typically decreases response times for all users [34]. It reduces overall system performance by making disk access more random, by making CPU and file caches less efficient, and so forth. To this end, we executed testing case a), depicted in figure 14, with database access to identify the system limits for optimal number of concurrent queries when using MySQL database as primary storage.

We observed that when we have more than two concurrent publication requests aiming to update same context, the system fails to satisfy 23% publication requests. The results for the first test condition are shown in Table 4. These results indicated significant increase in system performance as well as high failure rate. This lead us to do the test case a) again, when we grant publication request within the cache, as the primary storage, without having database access. Here, we stored the updated contexts offline, i.e. outside of the time-interval during which the query is executed, into the MySQL database (as the secondary storage). Based on the results depicted in

Figure 16 and Table 5, we determine that the large number of concurrent publication requests may also well be responded without any error by the system, when the system does not require database access in granting the concurrent requests. The measurements presented here were taken on local area network. One should also keep in mind that the large number of concurrent inquiry or publication requests is less likely to happen and exceptional cases in a real-life grid application deployed on a wide area network from the perspective of network latency, message delivery failures and data-loss.

# of simultaneous users	1	2	3	4	5
fail status	no fail	no fail	23% fail	30% fail	34% fail
Statistics (milliseconds)					
Maximum	67.871	497.097	646.622	563.133	621.713
*** Average**	**59.689**	**135.779**	**150.843**	**173.894**	**178.494**
Minimum	54.407	47.785	50.689	64.796	66.604
Standard Deviation	4.906	152.009	152.316	135.169	139.115

Table.4. Statistics for the condition where concurrent publishers send their requests to FTHPIS-WSContext Service with database access

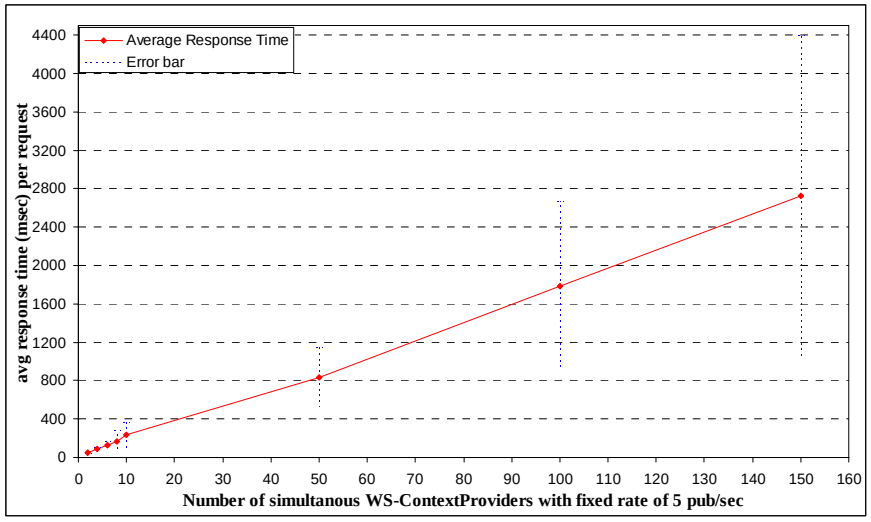


Fig.16. Average WS-Context Publication Response Time Chart - making 5 publication request/second by each context provider

# of simultaneous users	**2**	**4**	**6**	**8**
fail status	no fail	no fail	no fail	no fail
Statistics (milliseconds)				

Maximum	53.999	123.238	289.952	715.851
*** Average**	**46.844**	**88.391**	**122.864**	**170.131**
Minimum	40.867	46.097	47.361	65.008
Standard Deviation	4.459	21.009	53.344	114.401

# of simultaneous users	**10**	**50**	**100**	**150**
fail status	no fail	no fail	no fail	no fail
Statistics (milliseconds)				
Maximum	890.162	1960.656	4248.285	10880.92
*** Average**	**231.808**	**835.487**	**1788.542**	**2725.265**
Minimum	55.63	217.192	298.702	66.184
Standard Deviation	144.815	310.215	886.146	1675.578

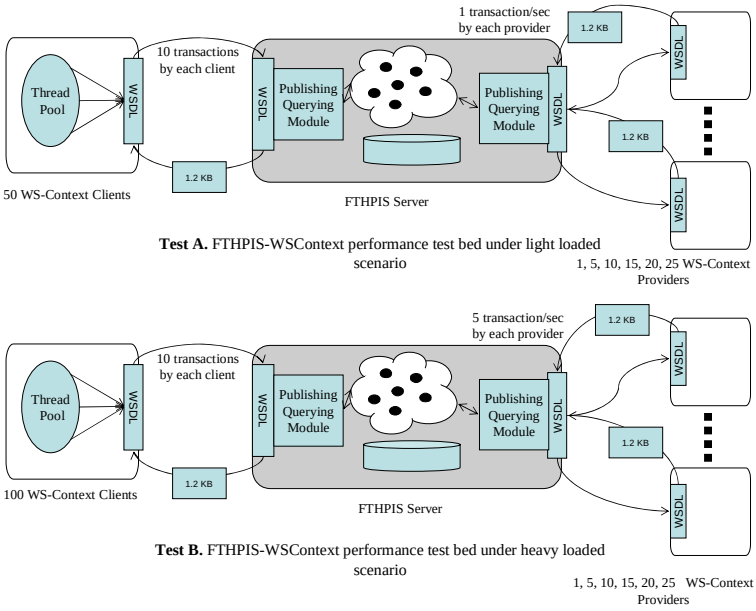
Table.5. Statistics of the experiment depicted in Figure 16. These measurements were taken with FTHPIS-WSContext Service without database access. Performance of the single provider is given in Table 2.

5.4. Experiment 3 –Experiment designed based on application use case scenarios:

The main goal of this experiment is to investigate how the system performs in a real application case scenario such as Patten Informatics workflow-style GIS application [1]. In this motivating scenario: GPS, Fault, and Seismic data bases, wrapped by OpenGIS web services, filtering and geo-processing services are distributed across various institutions. All these services interact with each other within a dynamically generic workflow to produce a common goal such as predicting an earthquake. The Pattern Informatics worklow-style grid application requires a session metadata manager to manage activities of the workflow. A session metadata manager is used to provide access/storage/search interface to metadata generated by the participating entities of the session. Here, we investigate the applicability of the FTHPIS-WSContext Information Service as a session metadata manager in Pattern Informatics domain.

In this motivating scenario, an example state-metadata might have information about the state of the workflow, such as “executing”, “completed” and so forth. We expect the size of shared state-metadata to be around 1.2 KB. We illustrate such context example in appendix A. As the session-state metadata is shared and highly updated, it has both multiple readers and writers. To this end, we expect concurrent publication requests as well as concurrent inquiry requests. We consider an example workflow session where there are numerous client web services (ranging from 50 to 100) polling information, while context provider web services (ranging from 1 to 25) publish the state changes of the workflow with varying frequencies.

We set up an environment where we have multiple-readers and multiple-writers which communicate through the FTHPIS system. We investigate the performance of the FTHPIS-Context Service implementation under both light and heavy loads with varying concurrent publication and inquiry requests. In this picture, there are WSContextProviders (corresponding to a context provider) and WSContextClients (corresponding to a context client) that have access to shared data containing statistics for up to 20 contexts where each context has multiple-reader/single writer access. The design of this experiment is depicted in Figure 17.



Test A. FTHPIS-WSContext performance test bed under light loaded scenario

Test B. FTHPIS-WSContext performance test bed under heavy loaded scenario

Fig.17. Experiments with varying number of concurrent clients and providers at various loads.

We performed two separate testing cases in this experiment: we measure the performance of the FTHPIS system from the WSContextClient perspective under a) light and b) heavy loads. For case a), **w**e look at the performance of the system with 50 query threads each issuing 10 queries to the FTHPIS node. We timed the complete round trip of all 500 queries issues by the WSContextClient for varying WSContextPublishers (1, 5, 10, 15, 20, and 25) each issuing one update requests per second.

For case b), we measure the system performance under heavier loads. We increase the number of querying threads to 100 and had each issue 10 queries. We also increased the frequency of updates to five per second for each WSContextPublisher. The aver-

age response time for light and heavy load with 1, 5, 10, 15, 20 and 25 publishers is presented in Figure 18.

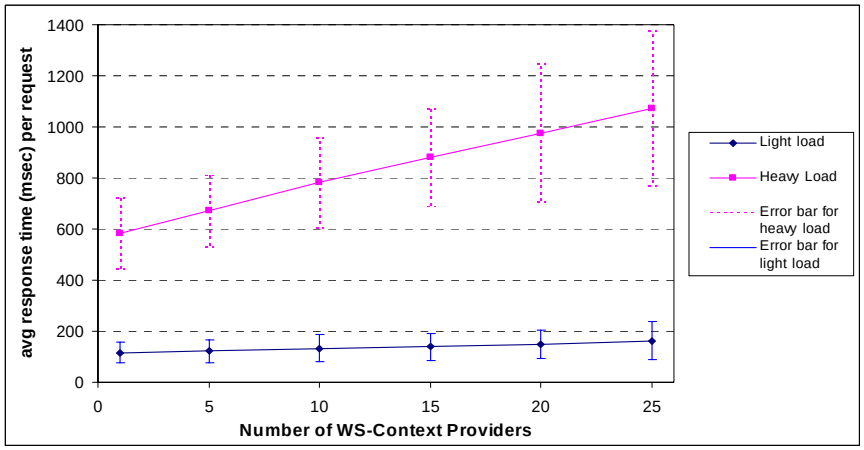


Fig.18. Average WSContext Client Round Trip Time (RTT) Chart for Light and Heavy Load

Results of the Experiment 3: As anticipated, the system showed better response times under light load compared with the system being under heavy load. It should be noted that the standard solution to multiple-reader/single-writer access at both levels of synchronization gives the readers priority. In severe cases the writes can suffer from starvation. We observe this reader-bias behavior in the results, as the number of querying clients increased, the system presents noticeable performance degradation. Based on the results, we conclude that the FTHPIS-WSContext Service is an applicable session manager service for work-flow style grid applications that are tolerable to a) average response times ranging between 100 and 160 milliseconds under light loads and b) average response times ranging between 0.5 to 1 seconds under heavy loads.

5.5. Experiment 4 –CPU thread scheduling latency experiment

In order to have better understanding of the system performance without effect of the time spent for CPU thread scheduling interference, we measure the actual CPU processing time for varying transactions. We performed two separate testing cases in this experiment: we measure the actual CPU processing time and average turn around time for a) inquiry and b) publication functions over varying transactions. Whereas in previous experiments we started our times just before sending off a query and stopping it once a complete response was received from the server, we now measured just the time necessary to query or write context into the server. Here, we wanted to determine what the actual performance of the system independent from network latency. We used a commercial profiler program called “OptimizeIt” (more at

www.borland.com/optimizeit) to measure the actual CPU processing time. We use the exact same testing case as it is depicted in test case a) in Figure 6 for inquiry function and again testing case a) in Figure 7 for publication function with the following exceptions. For case a) and b) we measure the average turn around time and CPU latency with 50, 200, 400, 600, 800 and 1000 transactions. In this experiment, we ran the FTHPIS server on the lightly loaded windows XP machine (kilimanjaro.ucs.indiana.edu), while the client application was running on cluster node-5. The results of the experiment for inquiry and publication functions are depicted in figures 19 and 20 respectively.

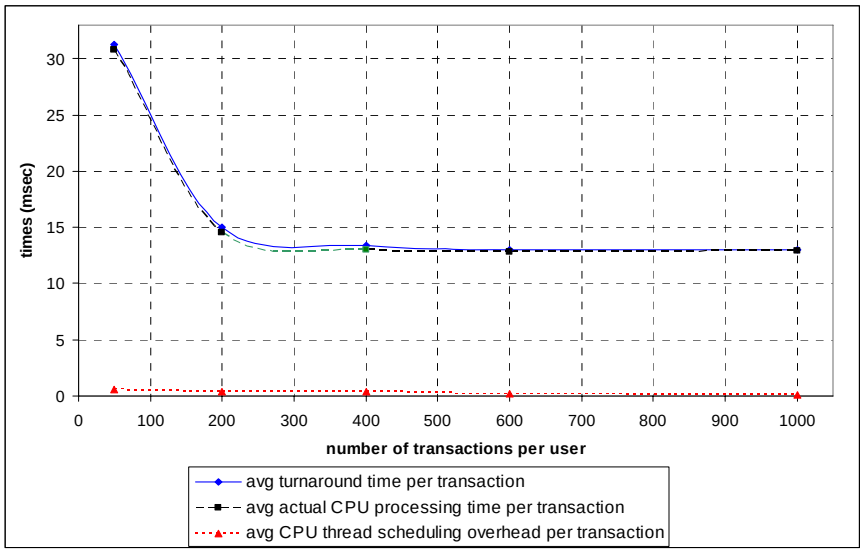


Fig.19. Measuring CPU Scheduling Latency spent in internal sub-activities of WS-Context inquiry function for 1 querying client under varying loads

Results of the Experiment 4: Based on the results, we observe two working modes: startup mode and initialized mode. The histogram of the average turnaround time, i.e. the time difference between the method entry and exit, is the sum of the other two histograms: actual CPU processing time and CPU thread scheduling latency. We note that the CPU thread latency is not an actual overhead for inquiry function of the system and the latency decreases in average as the number of transactions get increased. However, we observe a noticeable startup CPU scheduling latency (8.5 milliseconds) for the publication function. We also note that the CPU latency decreases in average for publication function as the number of transactions get increased. We conclude that CPU latency impact factor on system performance might be negligible if the system is in initialized mode.

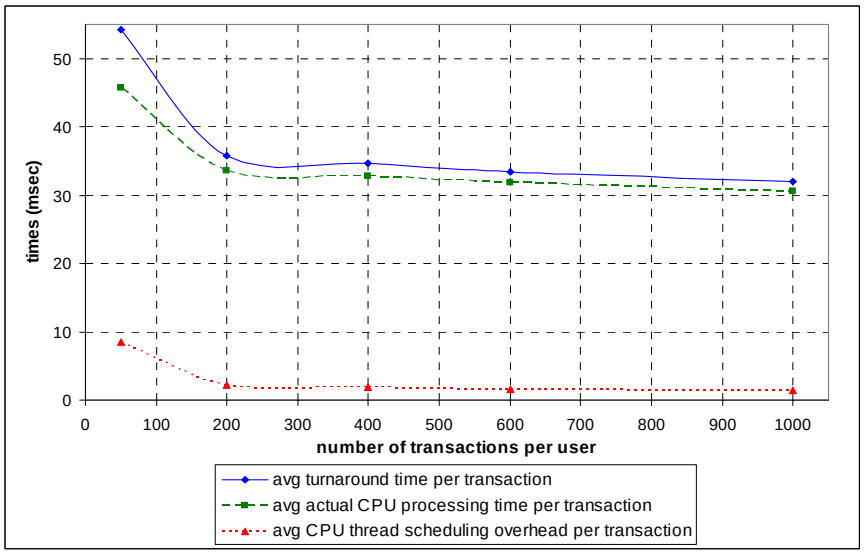


Fig.20. Measuring CPU Scheduling Latency spent in internal sub-activities of WS-Context publication function for 1 publishing client under varying loads

6 Conclusions and Future Work

In this paper, we have identified an important gap in Information Services for Grids that is lack of support for dynamic information in dynamically assembled traditional and Semantic Grids. We have presented an architecture that addresses key issues of managing dynamic metadata such as a) providing an efficient metadata access and storage methodology by taking into account changes in user demands and b) providing a P2P approach for access/storage request distribution among the peers of the system to capture the dynamic behavior both in metadata and the network topology. We perform an extensive set of experiments to evaluate the performance of the centralized version of the FTHPIS-WSContext Information. The performance results show the FTHPIS architecture can provide performance improvement over 18% for inquiry function and 22% for publication function by employing an expeditor module in its internal architecture. The promising low response latency results of experimental study on responsiveness indicates that high performance service conversation can be achieved with centralized metadata strategies with metadata coming from more than two services as opposed to service conversation with metadata only from the two services that exchange metadata. In addition, the performance indicates that efficient mediator services also allow us to perform collective operations such as queries on subsets of all available metadata.

The experimental studies on sustainability of the system shows that the large number of concurrent operations may well be responded without any error by the system. By comparing the results form studies conducted on latencies, we determine that CPU thread scheduling latency impact factor on system performance might be negligible if the system is in initialized mode, while the network latency may have a significant impact on the centralized version of FTHPIS-WSContext system performance.

We have discussed status of our implementation and report performance results from a prototype that is applied to sensor and collaboration grids.

Work remains to further develop a distributed metadata hosting environment by employing novel dynamic replication techniques and to evaluate the system as whole through extensive performance tests.

Acknowledgement: This work is supported by the Advanced Information Systems Technology Program of NASA's Earth-Sun System Technology Office.

References

1. Galip Aydin, Mehmet S. Aktas, Geoffrey C. Fox, Harshawardhan Gadgil, Marlon Pierce, Ahmet Sayar. SERVOGrid Complexity Computational Environments(CCE) Integrated Performance Analysis, Accepted as poster and short paper in Grid2005, Seattle, USA
2. H. Zhuge. Semantic Grid: Scientific Issues, Infrastructure, and Methodology, Communications of the ACM. 48 (4) (2005)117-119.
3. Wenjun Wu, Geoffrey Fox, Hasan Bulut, Ahmet Uyar, Harun Altay "Design and Implementation of A Collaboration Web-services system", Journal of Neural, Parallel & Scientific Computations (NPSC), Volume 12, 2004.
4. B. Plale, P. Dinda, 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.
5. Monitoring & Discovery System (MDS4) Web Site is available at http://www.globus.org-/toolkit/mds
6. A. Cooke, A.Gray, 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.
7. Ratnasamy, Sylvia et al. A Scalable Content-Addressable Network. Proc. ACM SIGCOMM, pp 161-172, August 2001.
8. Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. IEEE/ACM Trans. on Networking, 11 (1): 17-32, February 2003.
9. Serafeim Zanikolas and Rizos Sakellariou. A Taxonomy of Grid Monitoring Systems. Future Generation Computer Systems, 21(1), January 2005, pp. 163--188.
10. V. Dialani. UDDI-M Version 1.0 API Specification. University of Southampton – UK. 02.
11. Ali ShaikhAli, Omer Rana, Rashid Al-Ali and David W. Walker. UDDIe: An Extended Registry for Web Services. Proceedings of the Service Oriented Computing: Models, Architectures and Applications, SAINT-2003 IEEE Computer Society Press. Orlando Florida, USA, January 2003.

12. Simon Miles, Juri Papay, Terry Payne, Keith Decker, and Luc Moreau. Towards a Protocol for the Attachment of Semantic Descriptions to Grid Services. In The Second European across Grids Conference, Nicosia, Cyprus, pages 10, January 2004.
13. Miles, S., Papay, J., Dialani, V., Luck, M., Decker, K., Payne, T., and Moreau, L. Personalized Grid Service Discovery. Nineteenth Annual UK Performance Engineering Workshop (UKPEW'03), University of Warwick, Coventry, England, 2003.
14. 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.
15. Sivansubramanian S., Szymaniak M., Pierre G., Steen M.V. Replication for Web Hosting Systems. ACM Computing Surveys. Vol. 6, No. 3, September 2004, pp. 291-334.
16. M. Rabinovich. Issues in Web Content Replication. Bulleting of the IEEE Computer Society Technical Committee on Data Engineering, 1998.
17. Dilley, J., Maggs, B., Parikh, J., Prokop, H., Sitaraman, R., and Weihl, B. Globally distributed content delivery. IEEE Internet Computing 6, 5 (Sept.), 50-58. 2002
18. M. Rabinovich, I. Rabinovich, R. Rajaraman, and A. Aggarwal. A Dynamic Object Replication and Migration Protocol for an Internet Hosting Service. Proc. 19th Int'l Conf. Distributed Computing Systems, pp. 101-113, June 1999.
19. P. Rodriguez, and S. Sibal. SPREAD: Scalable Platform for Reliable and Efficient Automated Distribution Computer Networks, vol. 33, nos. 1-6, pp. 33-49, June 2000.
20. Bunting, B., Chapman, M., Hurley, O., Little M,, Mischinkinky, J., Newcomer, E., Webber, J., and Swenson, K. Web Services Context (WS-Context), available from http://www.arjuna.com/library/specs/ws_caf_1-0/WS-CTX.pdf
21. Bellwood, T., Clement, L., and von Riegen, C. UDDI Version 3.0.1: UDDI Spec Technical Committee Specification. Available from http://uddi.org/pubs/uddi-v3.0.1-20031014.htm.
22. Mehmet S. Aktas, Galip Aydin, Geoffrey C. Fox, Harshawardhan Gadgil, Marlon Pierce, Ahmet Sayar, Information Services for Grid/Web Service Oriented Architecture (SOA) Based Geospatial Applications, Technical Report, June, 2005
23. Shrideep Pallickara and Geoffrey Fox 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, Rio Janeiro, Brazil June 2003. See also: http://www.naradabrokering.org
24. N. Carriero and D. Gelernter. Linda in Context. Commun. ACM, 32(4): 444-458, 1989.
25. Extended UDDI and Fault Tolerant and High Performance Context Service Research is available at http://www.opengrids.org
26. Mehmet S. Aktas, Geoffrey C. Fox and Marlon Pierce. Managing Dynamic Metadata as Context. The 2005 Istanbul International Computational Science and Engineering Conference (ICCSE2005), Istanbul, Turkey.
27. China National Grid Project Web Site is available at http://www.cngrid.org/en_index.htm
28. Deborah L. McGuinness and Frank van Harmelen , OWL Web Ontology Language Overview, Editors, W3C Recommendation, 10 February 2004, http://www.w3.org/TR/2004/REC-owl-features-20040210/ . Latest version available at http://www.w3.org/TR/owl-features/.
29. Graham Klyne and Jeremy J. Carroll, Resource Description Framework (RDF): Concepts and Abstract Syntax, W3C Recommendation, 10 February 2004, http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/ . Latest version available at http://www.w3.org/TR/rdf-concepts/.
30. H.Zhuge, The Knowledge Grid, World Scientific Publishing Co., 2004.
31. H.Zhuge, Semantics, Resource and Grid, Future Generation Computer Systems, 20(1)(2004) 1-5.

32. Milojicic, Dejan S., et al. Peer-to-Peer Computing. HP Labs Technical Report HPL-2002-57, 2002
33. Georgina Saez, Amy Sliva, M. Brian Blake, Web Services-Based Data Management: Evaluating the Performance of UDDI Registries, ICWS 2004: 830-831
34. MySQL 3.23, 4.0, 4.1 Reference Manual avalible at http://dev.mysql.com/doc/refman/-4.1/en/index.html
35. M. Gerndt, R. Wismüller, 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. White paper. Shaker Verlag, 2004.
36. Geoffrey Fox Grids of Grids of Simple Services for CISE Magazine July/August 2004
37. D.J. Watts and S.H. Strogatz., Collective Dynamics of Small-World Networks, Nature. 393:440. 1998.
38. R. Albert, H. Jeong and A. Barabasi., Diameter of the World Wide Web, Nature. 401:130. 1999.

Appendix:

A. Sample Context XML metadata:

```
<?xml version="1.0" encoding="UTF-8"?>
<uddi_wsctx:context
        xmlns:uddi_wsctx="http://WSCTX.services.axis.cgl/uddi_wsctx_schema"
        xmlns:wsctx="http://WSCTX.services.axis.cgl/wsctx_schema"
        xmlns:uddi_ext="http://uddi.services.axis.cgl/uddi_ext_schema"
        xmlns:uddi="http://uddi.services.axis.cgl/uddi_schema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <uddi_wsctx:contextKey>
        uuid:ABCCE800-AB35-11DA-A4FC-C80C5880CB18-1141445798958
        </uddi_wsctx:contextKey>
        <uddi_wsctx:sessionKey>
        uuid:ABCCE544-CX35-11EA-BVFC-C34C7789CB33-1414457987978
          </uddi_wsctx:sessionKey>
        <uddi:name>
        <value>
        context://GIS/PI/ABCCE544-CX35-11EA-BVFC-C34C7789CB33
        </value>
        </uddi:name>
        <uddi_wsctx:value>COMPLETED</uddi_wsctx:value>
        <uddi_wsctx:accessRightInfo>
        <uddi_wsctx:others>
                <uddi_wsctx:readAccess>true</uddi_wsctx:readAccess>
                <uddi_wsctx:writeAccess>false</uddi_wsctx:writeAccess>
        </uddi_wsctx:others>
        </uddi_wsctx:accessRightInfo>
        <uddi_ext:lease>
                <timeout>1000</timeout>
                <isInfinite>false</isInfinite>
        </uddi_ext:lease>
        <uddi_wsctx:version>1</uddi_wsctx:version>
</uddi_wsctx:context>
```

B. 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>
        uuid:12114460-B4B6-11DA-A1DD-C2341CB5D80D
        </serviceKey>
        <businessKey>
        uuid: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>
                uuid:129679F0-B4B6-11DA-A1DD-E719F6E12358
        </bindingKey>
        <serviceKey>
                uuid: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>
                        uuid:6D712AF0-4ADA-11DA-BC65-C767C07EBBEA
                </uddi:tModelKey>
                <uddi:keyName>ServiceCategory</uddi:keyName>
                <uddi:keyValue>GIS-WFS</uddi:keyValue>
        </uddi:keyedReference>
        <uddi:categoryBag>
</uddi:businessService>
```