A NAMING AND DIRECTORY SERVICE FOR PUBLISHER-SUBSCRIBER'S STATUS DISSEMINATION

By

PING JIANG

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

MAY 2004

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of PING JIANG find it satisfactory and recommend that it be accepted.

Chair

ACKNOWLEDGEMENT

I want to thank Dr. Hauser’s invaluable advice and help. He is a very knowledgeable teacher and a very nice friend.

I feel grateful to thank Ioanna, Harald and my committee members Dr. Bakken and Dr. Dyreson’s for their help in GridStat and XML.

I want to thank the WSU Graduate School and EECS department for supporting my study and research.

I also want to thank my parents, Matt, Weihang and all of my friends for supporting me and encouraging me. I feel I am very lucky to be able to study in WSU and to meet all of you. Your love and encouragement makes me want to be a better person. I will work hard and be a useful person to the society.

A NAMING AND DIRECTORY SERVICE FOR

PUBLISHER-SUBSCRIBER'S STATUS DISSEMINATION

Abstract

By Ping Jiang, M.S.
Washington State University
May 2004

Chair: Carl Hauser

My thesis is to design, implement and test GridStat Directory Service (GDS), a naming and directory service to support GridStat, a publisher-subscriber status dissemination model for monitoring the status of the power grid. GDS is an XML-based directory service with authentication, authorization and replication.

We design GDS after looking at the designs of X.500, LDAP and many specialized directory services. We adapted the general purpose directory service functionality to the needs of a publisher-subscriber status dissemination model. We used CORBA as the communication medium and XML as the message format between clients and servers on the network.

GDS is organized into a two-plane hierarchical network, data and command, with leaf nodes available for clients to connect. We tested GDS with up to 1,600 searchable entities on the network.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Aim

The aim of this research is to design, implement and test a secure, reliable and scalable directory service to support the GridStat project. GridStat Directory Service (GDS) is a XML-based directory service with authentication, authorization and replication. The GDS servers are arranged in a two-plane hierarchical network with automatic recovery in the event that a server fails. The two planes are the Data plane and the QoS management plane.

1.2 Introduction to GridStat

The electric power grids in North America are large complex interconnected systems. The dynamic nature of the power grid is subject to regional or system-wide disturbances. The recent New York City power outage accelerates the need for a wide-area monitoring system for the grid. It has been shown in the investigation of the event in New York, that if a mature monitoring system were in place, operators would have know the conditions causing the blackout were present, up to 8 hours before it collapsed.

The power system market is mature, and old mechanical relays with no monitoring capabilities are still widely used. Relays protect the system by cutting off current in the event of a fault or an unusual condition on the line. Because mechanical relay technology was invented in the 1930's, the features available are extremely limited. When a fault in the system occurs, a mechanical relay will provide no helpful information to locate the fault. Transient problems will then almost never found, making the complete system less reliable. Additionally, existing automated control techniques guarding the stability of the system are local in scope. For example, modern power protection relays can take measurements on power lines locally connected to them, and detect possible problems with abnormal frequency or other related variables changes occurring on the connected line.

The newest technology commercially available will display line conditions from several PMUs (phasor measurement unit) locally and nation-wide using proprietary networks that are limited in application. However, the system for processing and analysis of the information is centralized. This creates a single point of failure for the system. Additionally, the scalability of centralized servers is limited. There are on the order of 150 million homes and offices in the US. As a future goal, a centralized system capable of monitoring all the lines in the US, or even one region, will require far more performance then is available. Reliability is also a key issue in this scenario. The potential failure of the central server leads to the failure of the whole measurement system. Thus, as described in [27], an efficient distributed measurement system would be extremely useful.

GridStat, [25], is a wide-area status dissemination service for the electric power grid designed to overcome the limitations of the current electric power communication infrastructure. It creates the possibility of decentralizing the control center so that functions can be put in different places depending on where it is needed.

Figure 1 from [25] shows the communication infrastructure and elements in the GridStat network. A status variable represents some device or phenomenon that has a state that changes over time. Publishers are the sources and subscribers are the destinations of status variables in the publisher-subscriber model. They connect to the communication infrastructure through edge status routers. Status routers make up the communication infrastructure. They are organized into a set of peer-to-peer networks called clouds, which are managed by a broker hierarchy. They behave as smart routers, receiving and forwarding status messages. Edge status routers allow publishers and subscribers to directly connect to them, thus providing the access points to the communication infrastructure.

A GridStat network is composed of two planes: a data plane and a QoS management plane. The QoS management plane contains the QoS brokers and is connected to the data-passing plane via the leaf QoS brokers. Each leaf QoS broker is in charge of one cloud in the data plane. The data plane contains the status routers and is directly connected to the publishers and the subscribers via the Edge Status routers. The Border Status routers connect different clouds together.

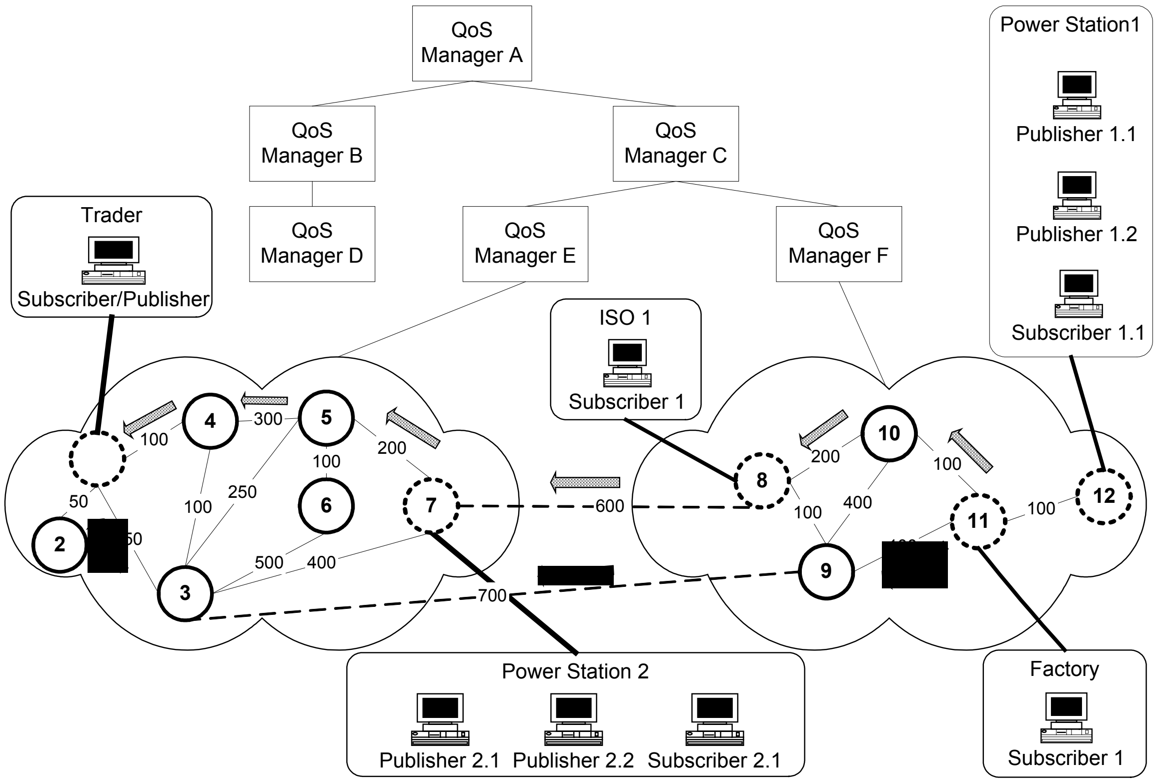


Figure 1.1: GridStat Network Hierarchy

Given a publisher and subscriber, the GridStat network can find the shortest path between the two, and set up connection.

1.3 Motivation for GDS (GridStat Directory Service)

In the current GridStat design, names and attributes of the publishers must be known before gaining access to their published data. The naming format is the fully qualified path from the root QoS broker via internal QoS brokers through leaf QoS brokers and to the subscribers/Publishers. For example, publisher 2.1 is named as A/C/E/Publisher 2.1 and subscriber 1 is named as A/C/F/Subscriber 1.

The current naming mechanism is convenient in locating elements in the system at the early stage of development in GridStat. In large real-life applications, as a security and management concern, clients who subscribe on the GridStat network shouldn't need to have prior knowledge of the structure of the system. It would be desirable to have a better naming mechanism.

In the case that a subscriber does not know which specific publisher's information it needs, it would be useful to be able to ask the GridStat directory service for publishers based on certain criteria. As a generic system, the GridStat directory service should provide common functionality of a directory service.

The GridStat directory service serves many different subscribers and publishers. To perform a look up or search, the communication between clients should be generic and easy to interpret. XML-based messages will provide these attributes; therefore, the directory service will use XML messages.

Stability is a critical issue in the GridStat network because there is no replication and recovery mechanism provided so far. It would be desirable for GDS to provide such a mechanism that can be used in the GridStat QoS management plane.

This thesis describes the design and implementation of an XML-based directory service with authentication, authorization, replication and auto-recovery.

1.4 Chapter layout

Chapter 2 contains a fundamental definition of a general-purpose directory. It also gives an overview of different directory services and the reason why they do not appropriately fit the needs of the GridStat network. Chapter 3 covers the design of our special-purpose directory service, GDS. It includes a description of naming module, communication infrastructure, and important protocols. Security and reliability considerations are analyzed in chapter 4 and chapter 5 and possible solutions will be provided. Chapter 6 gives an overview of an implementation based on the GDS design given in Chapter 3, 4 and 5. In Chapters 7, a performance analysis of GDS is presented. Chapter 8 gives a summary of the work done and a perspective on possible future work.

Chapter 2
GDS Requirements and Related work

2.1 General Directory Definition

The X.500 standard gives the following definition of a directory [17]: “The Directory is a collection of open systems which cooperate to hold a logical database of information about a set of objects in the real world.” But a directory differs from a database because it is organized in an object-oriented and hierarchical way.

The GDS directory service has functionalities similar to white pages and yellow pages in telephone books. A white page directory service provides the ability to look up in a dictionary a certain object using the name as an index while a yellow page directory service provides the means to find someone or some objects by the services they offer or other attributes.

2.2 GDS Requirements

GDS is based client-server communication. The clients are publishers and subscribers in the GridStat network. The servers are GridStat QoS managers in the QoS management plane.

It is a special-purpose directory service for GridStat network, thus it has many requirements.

- GDS is designed to be integrated into GridStat network by adding interface to QoS managers. An appropriate design decision should not add functionalities to status routers because they are designed to be light weight and route message fast.
- GDS should inherit the distributed natural of GridStat network.
- In GDS architecture, a client can only talk to the leaf QoS manager of the cloud it connects to. It can not access other leaf QoS managers outside the cloud.
- A GDS entry contains the information of a publisher's status variable. This information varies among different publishers. As a generic directory service for GridStat, GDS should be able to hold user defined attributes.
- GDS defines both client-server and server-server communication protocols. The client-server communication protocol should be easy for the clients to interpret, while the server-server protocol should be simple and fast for delivering.
- We assume in the future GDS servers will be put inside a dedicated network such as a virtual private network (VPN). Though there is still vulnerability in this type of network, they are relatively secure. Since it is not practical to let clients join the

dedicated network, in this thesis, we focus on the security between client-server communication.

- GridStat communication infrastructure is built using CORBA, this should also be the middleware on which GDS is built.

2.3 Directory services examples

There are many successful directory services available nowadays. By looking at some of them, we gain valuable insights for GDS.

2.3.1 Whois

The Whois [26] directory service is based on the client/server model. The server and database are running on a few specific central machines in the Network Information Center (NIC) and are query/response based. With the exponential growth of the Internet, a centralized Whois database has proven to be very inefficient. This necessitated a decentralized approach to storing and retrieving data.

The later version of Whois, Whois++ directory service [7] organizes the Whois servers into a distributed directory mesh that consists of index servers and base-level servers. Each layer of the mesh has a set of ‘forward knowledge’ (centroid), which indicates the contents of the various servers at the next lower layer of the mesh. The centroid generated by a certain server contains a list of attributes and templates along with the values of that server and is gathered by upper layer index servers. Whois++ also allows index servers to exchange information about sources they index, and to forward queries they receive to other knowledgeable index servers.

The next generation of Whois is Rwhois (Referral Whois), [39], a protocol which extends and enhances the Whois concept in a hierarchical and scalable fashion. It routes a user's query to the closest server that can answer the query. It focuses on the distribution of "network objects"--the data representing Internet resources or people--and uses the inherently hierarchical nature of these network objects (domain names, Internet Protocol networks, email addresses) to more accurately discover the requested information.

Whois and its successors are simple protocols for Internet-accessible white pages. There are no restrictions on the design of the client, provided it is passing queries to the server in the proper format [18]. However, its lack of versatile data representation would make it difficult to describe data entries required by GDS. Furthermore, Whois clients can query the servers, but not modify the information. The modifications are only performed locally at the servers. It is also designed to be a publicly accessible directory service, the security requirement is much weaker than GDS. Additionally, its mesh hierarchy does not fit the GridStat servers' tree hierarchy. For these reasons, we don't consider its use.

2.3.2 Domain Name System (DNS)

DNS, [12], provides the mapping from the host to Internet Protocol (IP) address in the Internet. According to [33], new IP version IPv6 can also be represented in DNS. After registering itself to the NIC organization, each domain is identified by a primary and (perhaps) secondary name severs. The name sever is responsible for maintaining the host name to IP address mapping for every host in its domain. We can explicitly determine the address associated with the given host name or find the host name based on its IP address by going though each related name server.

DNS security extension [33, 11] provides data integrity and authentication to security aware servers through the use of cryptographic digital signatures (SIG) that are included in the secure resource records (RR) with DNS names. There is a public key distribution service that stores and distributes authenticated public keys.

DNS maintains the name service in a distributed fashion and provides fast lookup in the namespace. But it also has some limitations that make it not suitable for GDS. Part of the design philosophy of DNS is for the data in it to be public and that DNS gives the same answers to all inquirers. Following this philosophy, even though there is a DNS security extension; it is only designed for individual security aware resolvers. DNS serves a limited range of information, the most important being the mapping between host names and IP addresses. And the search capability is very limited, thus DNS cannot provide rich directory service functionality.

2.3.3 NIS/NIS+

NIS/NIS+, [16] is Sun's remote procedure call (RPC)-based operating system directory. It provides generic database access facilities that can be used to distribute common configuration information such as password and hostname to all hosts on the network within a NIS domain. A domain has one master server that has the canonical version of information and several slave servers that have copies of the information. The security issue is very important in the NIS/NIS+ system, because an exposed NIS server can provide hackers with huge amounts of information about the network, including a logical map of the machines on the network and a list of user accounts.

NIS/NIS+ naming services store information in a central place that users, workstations, and applications must have to communicate across the network. It is usually designed for workstations in a local area network (LAN). The centralized nature of NIS/NIS+ is not well suited for GridStat's requirement.

2.3.4 CORBA Naming Service

The Naming Service is a standard service for CORBA [30] applications. It associates names with object references in its name server, thus allows clients to find objects by looking up the corresponding names. Bind operation adds an entry to the name server and Unbind operation deletes one. However, CORBA Naming Service is a centralized system which has the potential scalability and reliability problem for a large system. And there is no security taking place for accessing it. Thus, it is not suitable for GDS.

2.3.5 Jini lookup Service

The Jini lookup service [24] is similar to a traditional distributed system's naming service such as CORBA Naming Service: it is the place where clients go to find services. Services are registered in the lookup service by a serialized proxy object. A service has one or more attributes. It initially registers itself to the lookup service. A client uses a template specifying attributes to search the lookup service, and is returned one or more matched services.

There are some advanced features in the Jini lookup service. For example, a serialized proxy object is issued a lease after registration. It must renew the lease to sustain its presence. This mechanism maintains the availability of the services and self-healing of the lookup service. Another advanced feature of the Jini lookup service is that a service can be configured to be visible only to particular groups. This segregation allows practical administration. The suggested method for replication in the Jini lookup service is to start a second lookup service. The replica lookup service will automatically update itself to contain all the registered services in the original lookup service.

Jini is powerful lookup service, but it is not applicable to GDS. Services are different from each other by their Java types, not their names [5]. If we want to use Jini to differ between GDS entries, we have to make them have different java types. This is not practical. Furthermore, it provides only a yellow page service, not white page service.

2.3.6 Grapevine Registration Data Base

Grapevine [1], as one of the earliest location independent naming systems, provides facilities for the delivery of digital messages such as e-mails. It maintains a registration database that maps names to information about the users, machines, services, distribution lists, and access control list that those names signify. The registration database uses a partitioned naming scheme for each entry in the registration database with a two-level hierarchy: a local registry name and name globally unique within the registry. Because any message server can accept a message from any source for delivery, a message can be replicated along the delivery path. This achieves a highly available message delivery service. Grapevine allows a looser definition of consistency that results in higher availability of the update function. Following an update, it will eventually become consistent, if no further updates occur.

Adding or deleting an entry is complicated, because the Grapevine registry is replicated on every registration server. It has too much propagation delay. Thus we will not use it in GDS.

2.2.7 X.500

The general-purpose directory services X.500 and its successor LDAP provide a standard reference for directory service design.

X.500 is a standards-based protocol that is designed to build a distributed, global directory. Each Directory entry contains information about one object (i.e. a person), which is built from a collection of "attributes". X.500 hierarchically organizes the namespace in its Directory Information Base (DIB) in the Directory Information Tree (DIT). The entire DIB is distributed through the world-wide collection of DSAs (Directory System Agents, which hold a particular subset of the DIB) which form the Directory. The DSAs employ two techniques to allow this distribution to be transparent to the user, called "chaining" and "referral", [21]. When a query comes into a directory server and the server cannot satisfy the request directly, the server may have information about a different server that does indeed have the answer. Chaining is where the first server talks with the knowledgeable server while referral is where the first server refers the knowledgeable server to the client.

X.500 defines the basic structure and functionality of a directory service. It has decentralized maintenance, very powerful searching capabilities, and a flexible and extensible namespace.

X.500 uses Directory Access Protocol (DAP) as the protocol to connect client systems to the directory server. But DAP requires that clients use the Open Systems Interconnect (OSI) protocol stack as well as significant amount of memory and other system resources. This has made it unsuitable for most implementations. This has driven the need for LDAP, the Lightweight DAP.

2.2.8 LDAP

LDAPv3, [19], [20], as a subset of DAP, is a simplified version of X.500, It eliminates seldom-used functionality of DAP. It runs directly over TCP bypassing session and presentation layers of OSI. LDAP servers use fewer command functions to provide roughly the same level of functionality as X.500. It reduces the burden of the directory service client by simplifying the encoding process used to transmit the distinguished names of directory entries.

A client initiates a connection with the LDAP server by sending the server a "bind" operation that contains the authentication information. LDAP authorization identity request and response control are defined in [37]. A client can submit an authorization identity request as part of a bind request. If it succeeded and resulted in an identity (authzId), the authzId will be returned in a bind response. LDAP Replication as described in [34] is a process where a directory is replicated onto the local machine to enable access to data in case if there is no server connection. Each time the replica does a query on the LDAP server, it will get all the added, deleted and modified records since the last replication time.

X.500 and LDAP are general-purpose directory services. To apply their technique in GDS, there are some important issue.

As stated in the GDS requirement section, a GDS client can only talk with its own leaf server. Thus referral is not supported in GDS. The search continuation is done only by server communication.

LDAP defines the following operations: search, add, delete, modify, modify RDN, bind, unbind, and abandon. In GDS, add, delete, search, modify, bind, unbind and search are supported. The modify RDN operation is not used, because modify RDN can be done by the modify operation. Currently, the abandon operation is not defined. In addition, another operation named FindPath is used. This is because one of the major purposes of GDS is to quickly find the path for a given name.

2.3 XML

Extensible Markup Language (XML), [15], is a simple, very flexible text format originally designed to meet the challenges of large-scale electronic publishing. The structure of XML files are described using Document Type Definition (DTD) or XML Schema. Both of them are self-descriptive. XML is playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere.

2.3.1 XDAP

Extensible Directory Access Protocol (XDAP), [13], specified in XML, describes an application layer client-server protocol for a framework representing the query and result operations of directory services. XDAP is a specification for frameworks with which these directory namespaces can be defined, used, and in some cases interoperate. These frameworks allow a directory namespace to define its own structure for naming, entities, queries, etc.

XDAP is a directory access protocol, not a directory service. It only supports requests and responses for authentication, session and service inquiry, and directory search. It does not provide a naming module or replication mechanism. XDAP provides only a simple client authentication mechanism. The simple client authentication mechanism uses a variant of the PLAIN SASL mechanism described in [8], where it provides the specification of an authorization identifier, authentication identifier, and password as a single string separated by ASCII NUL characters. A passive attack is sufficient to recover client's identifiers and passwords. Because XDAP does not provide enough directory access functionalities and security, it is not suitable to be used in GDS.

Because XML is a simple text format and is easy to interpret, it is a desirable choice to describe the directory access protocol.

2.3.2 XML and Java conversion tools

There are many XML and Java conversion tools. Breeze XML Binder, [6], is a tool produced by Breeze Factor. Castor, [9], is an open source project under ExoLab. JAXB Reference Implementation, [14] is created by SUN. And XGen, [42], is a tool produced by Commerce One. A comparison of them is provided at [28]. These tools generate Java classes from instances of XML Schema in order to represent the structures defined therein. The generated code has the ability to convert from XML format to Java objects and vice versa.

2.4 Web Services

In general, web services are services offered by one application to other applications via the World Wide Web [41]. Web services and consumers of web services are typically businesses. The provider of web services can also be the consumer of other web services. Because web services depend on the ability of parties to communicate with each other using different information systems and computing platforms, XML and Java platform play a central role in the web services.

The similarity between web services and GDS is using XML-based messages in the client-server communication. This makes it easier to port GDS into a web services provider. The way to enable web services is described in [40].

Web services functionality would be useful because it provides standardized user interface. However the predominant transactions are business-to-business (B2B) [41]. Adding web services functionality will increase the complexity of GDS. Due to the aim of GDS is to provide a directory service, in the currently implementation of GDS, we didn't make GDS a web service provider.

2.5 CORBA

The Common Object Request Broker Architecture (CORBA), [30], is an open distributed infrastructure being standardized by the Object Management Group. CORBA specifies a system that provides interoperability between objects in a heterogeneous, distributed environment and in a way transparent to the programmer. It can handle a large number of clients, at high hit rates, with high reliability. GDS is built using CORBA for communications. This is also a direct result of GridStat using CORBA.

2.6 Conclusion

GDS is a special-purpose directory service for the GridStat network. It is a distributed system that needs more scalability and reliability than centralized systems such as CORBA Naming Service, Jini and NIS/NIS+. It has a wider coverage of security requirements than DNS, because authentication and authorization are required mechanism all the server-client communication. Besides structural difference, it also requires more efficiency than Grapevine and richer data representation than Whois.

GDS synthesizes concepts from other, established directory service examples as above.

The GDS protocol and architecture derive a great deal of structure from the Domain Name System (DNS) and borrow directory service concepts from other directory service efforts, primarily LDAP.

To be easily interpreted by the clients, GDS uses an XML-based directory access protocol. XML files are self-descriptive and human-readable. Future GDS users can develop their own software to access GDS easily.

Chapter 3
GDS Design

3.1 GDS Structure

GridStat Directory Service provides new features to the GridStat network. The design must be compatible with the current GridStat structure. QoS brokers become GDS servers by adding new interfaces to them, which will be described later. The hierarchy design is Figure 3.1.

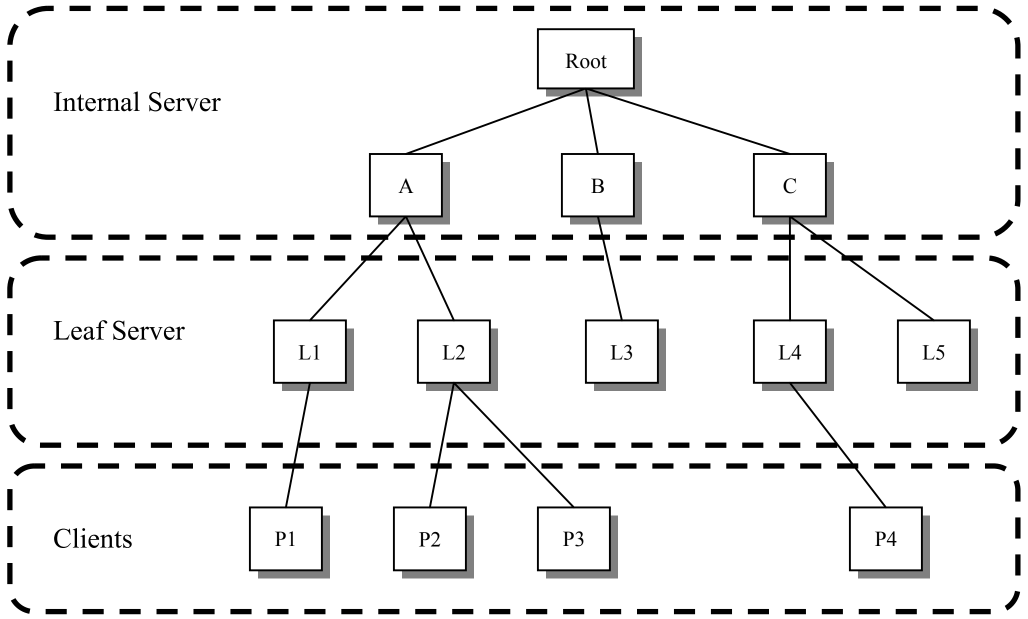


Figure 3.1: GDS Structure

Only the leaf QoS brokers store the directory entries. This is designed to distribute the internal and root QoS brokers burden and make the system more reliable. Each directory entry has the information of the publisher and its publishing item. The combination of the two attributes must be distinguished from other entries. Thus, the same publisher will have several entries in the directory if it publishes multiple items. The internal QoS brokers connect to the leaf QoS brokers to form a tree. The internal QoS brokers don't contain directory entry information; they only connect the leaf QoS brokers together and route request to the leaf QoS brokers and other internal QoS brokers.

We call both the GridStat publishers and subscribers the clients in the directory service; the QoS brokers are called the servers. Due to the design of GridStat network, the communication between the publishers/subscribers and QoS brokers must go through the Edge Status Routers.

In the following parts of this thesis, the root GDS server is a service of the GridStat root QoS broker; internal GDS servers are a service of the GridStat internal QoS brokers; leaf GDS servers are the GridStat leaf QoS brokers. Clients are part of each GridStat publishers and subscribers. The QoS hierarchy becomes a GDS tree. All the GDS servers who belong to the same parent are called siblings.

3.2 Naming

The GridStat network needs a fully qualified path from the root QoS broker to the client to locate it. This has been convenient in the early stage of GridStat development, but as it matures, a more scalable mechanism is needed.

GDS provides separation between the name and the path of a client. All servers' and publishers' names in GDS are arranged in a hierarchical manner—forming the Directory Information Tree (DIT) (see Figure 3.2). The other type of client, subscriber, does not need a name in GDS. Each entry in a directory is identified by its Distinguished Name (DN). It is composed by concatenating the entry's Relative Distinguished Name (RDN) with those of its superiors along the path to root entry. An RDN consists of an attribute name, the equal sign and the attribute value (e.g. "pn=publisher1"). RDNs must be different for all siblings in DIT. This ensures that all entries will have a unique DN. The attribute used in the RDN is called a naming attribute. Currently the internal QoS brokers don't have any geographic or national regions, thus the attribute name 'dc' —short for domain component—is used. The 'lf' is the attribute name of a leaf broker and 'pn' means publisher name.

From a publisher's DN, it is very easy to know the path. For example, "publisher1" with DN "dc = root, dc = internal2, lf = leaf2, pn = publisher1" has path root/internal2/leaf2/publisher1. Each leaf GDS server has a record of its RDN and DN which is learned in the formation stage of the directory server tree.

The naming module provides flexibility not present in the physical-based path names. For example, we can find publisher with RDN 'publisher1' under leaf server 'leaf2'.

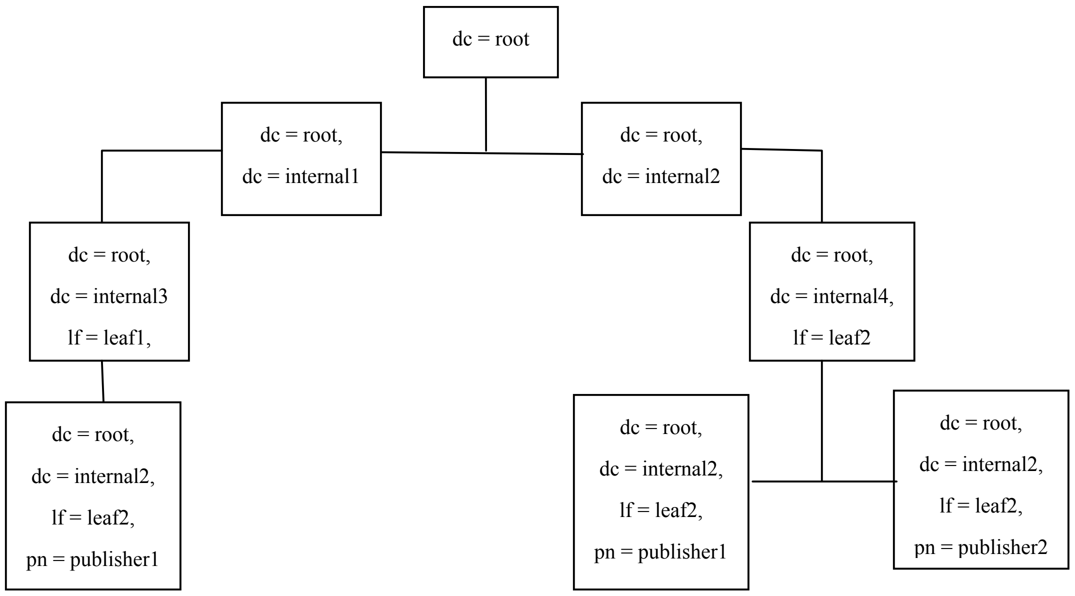

dc = root
dc = root, dc = internal1
dc = root, dc = internal2
dc = root, dc = internal3 lf = leaf1,
dc = root, dc = internal4, lf = leaf2
dc = root, dc = internal2, lf = leaf2, pn = publisher1
dc = root, dc = internal2, lf = leaf2, pn = publisher1
dc = root, dc = internal2, lf = leaf2, pn = publisher2

Figure 3.2: Namespace example

3.3 XML-based Directory Access Protocol

This protocol is designed to provide Bind, Unbind, Add, Delete, Modify, FindPath and Search operations. Each operation, except Unbind, includes a request from a client to a server, and its corresponding response from the server to the client. We use an XML schema to describe the message format. The schema includes the collection of attribute type definitions and objects class definitions. To make the schema readable, we arrange it in a precise manner. The whole XML schema is in appendix A, and the XML schema notation is described in [43].

Clients can only talk with the server within a connection session. A connection session starts after a client sends a Bind request and is authenticated. All the requests except Unbind request will receive corresponding responses. A client can then perform different operations. Given RDN(s), FindPath finds the path of the matching publishers whose DN contains the RDN(s). Only publishers have the right to perform Add, Delete and Modify operations. A connection ends when an Unbind request is received from the client or when the connection session is timeout. A connection session can be extended if another Bind request is received during this session.

3.3.1 Message

Uniform message format exchanging between client and server is defined in Figure 3.3.

```
Message ::= SEQUENCE {
       long      MessageID,
      sring      MessageType,
       OP     CHOICE {
                 bindRequest       BindRequest,
                 bindResponse      BindResponse,
                 unbindRequest     UnbindRequest,
                 addRequest        AddRequest,
                 addResponse       AddResponse,
                 deleteRequest     DeleteRequest,
                 deleteResponse    DeleteResponse,
                 pathRequest       PathRequest,
                 pathResponse      PathResponse,
                 searchRequest     SearchRequest,
                 searchResponse    SearchResponse,
                 modifyRequest     ModifyRequest,
                 modifyResponse    ModifyResponse
       }
}
```

Figure 3.3: Message

- MessageID: Unique long number to distinguish between messages from one client. This is very useful when a server is processing several different requests from the same client.
- MessageType: It tells which choice to use from the choice collection.

A client sends a request encapsulated in a message with a unique MessageID. The server eventually echoes a response with the same MessageID corresponding to the request. The message type indicates the choice of the message. For the CHOICE field, only one operation can be selected.

A message represents either a request or a response. A ClientName attribute is included in every request message. It is for the server to identify and authorize the client.

3.3.2 Bind

A Bind operation initiates a connection session from a client to its leaf QoS broker (a leaf server in GDS), and to allow the authentication of the client to the server. It must be the first operation received by a server from a client in a connection.

```
BindRequest ::=    SEQUENCE {
        name                    ClientName,
        authentication          CHOICE {
                                Anonymous          [0],
                                Message Digest         [1],
        }
        string         Credentials
}
```

Figure 3.4: Bind Request

- ClientName: the textual name of the client which sent the request
- Authentication: choice of authentication method
- Credentials: information used to authenticate the name, if any.

Upon receipt of a Bind request, a server will authenticate the client based on the

Authentication choice and Credentials. The server will send a Bind Response to the client indicating success or failure or need more information.

Bind Response

After receiving and processing the client Bind request, the server will then send a Bind response back to the client indicating if the bind operation is successful or not.

```
BindResponse ::=  ENUMERATE {
       Success                 [0],
       MethodNotSupport        [1],
       StrongAuthRequired      [2],
       WrongCredentials        [3],
       SeverUnavailable        [4],
       Other                   [5]
}
```

Figure 3.5: Bind Response

- Success: the Bind request is successfully processed, and the client is authenticated to connect to the server.
- MethodNotSupport: the authentication method is not supported.
- StrongAuthRequired: the bind to the server, client needs stronger authentication method.
- WrongCredentials: incorrect password or credentials that the server cannot use to

process the authentication.

- ServerUnavailable: the server is disconnect or shutting down.

This authentication is only between a client and its own leaf GDS server. Other access control issue should be considered later.

3.3.3 Unbind

A client can terminate a connection by issuing the unbind request.

```
UnbindRequest ::= NULL
```

Figure 3.6: Unbind Request

There are no parameters in the Unbind request message.

The client sends an Unbind request to the server if the operation requested by the client is completed, timeout or terminated by errors. After sending the Unbind request, the client assumes that the connection is terminated. Thus the server doesn't send any response message to the client, and it discards all the outstanding requests of this client.

Standard Response

Before we proceed to define other messages, we first analyze the common server responses.

- Success: This means a request is processed successfully by the server. This doesn't indicate that a FindPath or Search request finds matching entries.
- MessageInvalid: When a message is not validated to be a correct XML file based on the schema, it is invalid.
- NotBound: If a client is not bound to the server, none of its other requests can be processed.

We define a standard response message called StdResponse to represent these common responses. It may be used directly or integrated into other response messages.

```
StdResponse ::= Enumeration {
       Success              [0],
       MessageInvalid       [1],
       NotBound             [2],
       UnknownError         [3]
}
```

Figure 3.7: Standard Response

3.3.4 Add, Delete, Modify

Add, Delete and Modify operations can only be performed by publishers. Only the client and its leaf GDS server are involved in these operations. No further communication between the leaf server and other GDS servers is needed.

Each Add, Modify or Delete message contains both “PublisherName” and “PublishItemName” attributes as they are used to identify a specific entry in the directory.

3.3.4.1 Add Operation

The Add operation allows the client to add status events and their properties to its own leaf GDS server. When a publisher connects to the GridStat network or starts to publish new status events, it adds the new status event information to the leaf GDS server of its domain as new entries by using Add operations.

```
AddRequest ::= SEQUENCE {
        name                ClientName,
        name                PublishItem,
        int                 PublishRate,
        optionalAttribute   OptionalAttributeList
}

OptionalAttributeList ::= SEQUENCE {
        string              AttributeName,
        attributeType       AttributeType,
        string              AttributeValue
}
```

Figure 3.8: Add Request

- PublishItem: name of the publish item.
- PublishRate: the rate of the publish item, unit mps (message per second)

- OptionalAttributesList: contains a sequence of user-defined attributes.

The server performs the Add request and sends the result back to the client in an Add response (Figure 3.9).

```
AddResponse ::= ENUMERATE {
       StdResponse Response,
       EntryExist,
       …
}
```

Figure 3.9: Add Response

The AddResponse is either from standard response or other response specially related to add operation.

3.3.4.2 Delete Operation

When a publisher leaves the network or stops publishing some status events, it then deletes the entries for those status events from the leaf GDS server by using a Delete operation (Figure 3.10).

```
DeleteRequest ::= SEQUENCE {
       name          ClientName,
       name          PubliserName,
       name          PublishItemName
}
```

Figure 3.10: Delete Request

The server performs the Delete request and sends the result back to the client in a Delete response (Figure 3.11).

```
DeleteResponse ::= ENUMERATE {
          StdResponse Response
          EntryNotExist,
          …
}
```

Figure 3.11: Delete Response

3.3.4.3 Modify Operation

The Modify operation allows the client to modify properties of certain status events in its own leaf GDS server.

```
ModifyRequest ::= SEQUENCE {
       name          ClientName,
       name          PublisherName,
       name          PublishItemName,
       modification  ModificationList
}

ModificationList ::= SEQUENCE {
       Operation     Eumerate {
                     Delete        [0],
                     Replace       [1]
       }
       name          AttributeName,
       type          AttributeType,
       value         AttributeValue
}
```

Figure 3.12: Modify Request

- ModificationList: contains a sequence of attributes that a client wants to modify. For each attribute, it can choose between two different operations.
- Delete: the server first matches the attribute's name, and then deletes this attribute from the entry including its name, type and value. If this operation is used, the attributeType and attributeValue are left blank.
- Replace: the server first finds a matching attribute name, then sets the type and value to the attributeType and attributeValue.
- attributeType: the new type of the attributed to replace the old one if applicable.

- attributedValue: the new value of the attribute to replace the old one if applicable.

Modify response (Figure 3.13) is the same as delete response.

```
DeleteResponse ::= ENUMERATE {
           StdResponse Response
           EntryNotExist,
           …
}
```

Figure 3.13: Modify Response

3.3.5 FindPath

One of the most important operations in GDS is to map names in one namespace to names in another (usually in a lower level). The FindPath operation allows clients to easily find the path(s) to the publishers by providing part of the publishers' DN. For example, "pn=publisher1" sets the searching scope of FindPath to the whole directory, whereas "lf=leaf1, pn=publisher1" searches only within leaf server "leaf1".

```
PathRequest ::= SEQUENCE {
       name              ClientName,
       nameGDS           PublisherName,
       int               MaxEntryReturn,
       int               Timeout
}
```

Figure 3.14: FindPath Request

- PublisherName: The DN or (RDNs) provided by the client to find a publisher's path.
- MaxEntryReturn: limits the maximum number of entries returned. There is a default number and a maximum number for it in order to prevent using too many GDS resources.
- TimeOut: TimeOut is the maximum time a search result can be accepted measured in milliseconds. After sending the request message, the client waits for at most TimeOut length of time, then it either received a response message or it'll discard any future responses for this request.

Upon receipt of the client's path request, the local leaf GDS server searches its own directory entries first. If the number of matches is less than MaxEntryReturn, FindPath will perform a search continuation. It forwards the request to its parent server. The parent server then talks to its children in parallel and asks them for results. It may also forward the request up to its parent if the search is not satisfied. A FindPath request is done when the request is satisfied (the number of entries found is equal to MaxEntryReturn), the request has timed out or the whole directory is searched.

The local leaf GDS server eventually returns all the final responses to the client.

```
PathResponse ::=  SEQUENCE{
       stdResponse          Response,
       name                 Publisher,
       path                 ClientPath
}
```

Figure 3.15: FindPath Response

- Publisher: The publisher's name is not needed in the response, but it makes the response clearer as all the returned paths refer to it.
- Path: There can more than one ClientPath returned. They must be in the format of ([a-zA-Z0-9]+/)*

3.3.6 Search Operation

The search operation is designed to add more of the yellow-page directory service functionalities to GDS. The client can look up a publisher's information and it can search publishers by different attributes or their combination. It operates in a similar to the FindPath operation.

```
SearchRequest ::= SEQUENCE{
       name                 ClientName,
       optionalAttributes   Criteria, (*)
       int                  maxEntryReturn,
       int                  TimeOut,
       filter               Filter, (*)
       name                 ReturnAttributeNameList (*)
}

filter ::= ENUMERATE {
       AND           [0] set of filters,
       OR            [1] set of filters,
       NOT           [2] filter,
       EQUAL         [3] assertion,
       GE            [4] assertion,
       LE            [5] assertion,
       SUBSTRING     [6] string
}
```

Figure 3.16: Search Request

- Filter: a Filter defines the matching rule of a search. A Filter can only return ‘TRUE’ or ‘FALSE’ value. An assertion contains a name and a value of a variable. For example, the criteria “attributeName = ‘Location’, attributeType = ‘String’, attributeName = ‘Pullman’”, an assertion would be the variable ‘Location’ with value ‘Pullman’. The attributeType is currently not used in matching criteria. ‘GE’ represents greater or equal and ‘LE’ represents less or equal. If the string described in the ‘SUBSTRING’ choice is present in the attribute value, then the filter returns true, otherwise false. This is only applicable when there is an attribute named "Description". 'AND' and 'OR' choices should have at least one filter. 'AND', 'OR' and 'NOT' can form combinations of filters.

- ReturnAttributeNameList: it specifies the attributes name that needs to be returned to the client. For a yellow-page search, the ReturnAttributeNameList needs only contain the publisher's name. If any attributeName in the list is not found in the entry or not recognizable by the server, the server will ignore this attribute.

```
SearchResponse ::=  SEQUENCE {
        attributes        ReturnAttributesList
}
```

Figure 3.17: Search Response

- ReturnAttributesList: The Search response will return all the attributes and their values described in the 'ReturnAttributeNameList' filed of Search request. If any 'attributeName' is not found or not recognizable by the server, it is ignored thus not shown in the 'ReturnAttributesList'.

3.4 Simple Server Communication Protocol

A path or search request may be forwarded by one leaf GDS server to another server.

We define a new protocol for this communication link. The reason we define another protocol instead of the XML-based Directory Access protocol is because:

- Only path, search request or response may need to be forwarded.
- We assume GDS servers are inside a dedicated network, and is relatively secure, thus no other security mechanism is currently taking place.
- The conversion between XML file and Java classes takes a certain amount of time.
- Only the developer for the GDS network needs to know this protocol.

Each message is encapsulated in a String and all variables in a message are separated by a whitespace character. The message type information is included at the beginning of the message. The type includes "PathRequest", "PathResponse", "SearchRequest", "SearchResponse". PathRequest and PathResponse is shown in Figure 3.18.

```
PathRequest                        PathResponse
   String {
      "PathRequest"                   String {
      PublisherName                   "PathResponse"
      RemainEntryReturn               NumberOfReturnPaths
   }                                  Path 1
                                      Path 2
                                      …
                                      Path n
                                      }
```

Figure 3.18: PathRequest and PathResponse in Servers' Communication

SearchRequest and SearchResponse is shown in Figure 3.19.

```
SearchRequest
   String {
      “SearchRequest”
      PublisherName
      Criteria
      ReturnAttrName 1
      ReturnAttrName 2
      …
      ReturnAttrName n
      RemainEntryReturn
   }
```

```
SearchResponse
   String {
   “SearchResponse”
   ReturnAttrName    1
   ReturnAttrValue   1
   ReturnAttrName    2
   ReturnAttrValue   2
   …
   ReturnAttrName    n
   ReturnAttrValue   n
   }
```

Figure 3.19: SearchRequest and SearchResponse in Servers’ Communication

Chapter 4
Security

Given the concern of the security issue of this critical infrastructure, GDS must be secure from external intrusions.

4.1 GDS Security Issue

There are two different types of clients in GDS. GDS provides a read-only white-pages service to the subscribers which should be publicly available. Because the publisher has write access to GDS, a malicious user may change the information in the directory.

Because GridStat QoS brokers are located in a dedicated network, we assume the communication within the GridStat QoS brokers plane is secure. The data-passing plane of GridStat that provides the communication platform for the XML directory access protocol is connection-oriented. Though we assume direct connection between Edge Status Routers and leaf QoS brokers, there are many inter-mediate status routers. As an integrity concern, the data on its way between clients and servers may be changed in transit. Meanwhile as a privacy concern, an unauthorized person (man in the middle) may overhear protected data.

Common techniques that enhance the security include authentication and authorization. And to ensure integrity protection and privacy protection in an insecure environment, security layers have to be deployed. Security layers are commonly provided at the transport layer. The relevant standards in this field is the Secure Socket Layer (SSL) [2]. Since CORBA hides the socket layer from the user, we could not directory apply SSL. But the idea of SSL that uses encryption and decryption between two ends of a connection can be deployed in GDS.

4.2 Authentication

Authentication is the process where one party supplies to a requesting party information that identifies itself. This information guarantees that the originator is not an imposter.

In GDS, the first challenge is Authentication. Authentication is tremendously important to the clients (Publishers) which have directly write access to the directory entries.

In a password-based scheme, the client prompts the user for his or her name and password. The client relays this information to the server. The server checks the name and password and either allows the user into the system or denies access. The obvious solution is to send the user's name and password directly to the server. This is not a very secure solution.

The major disadvantage of plain text passwords is that they are easily to compromise. Anyone who can physically eavesdrop on the communication between client and server may learn the password. Once obtained, the credentials can be reused until the password is changed. The attacker might even change the password himself and thus lockout it’s original holder.

Great care must also be taken to protect the passwords stored on the server. Anybody who is able to read these secrets will get instantaneous access to all entries in the directory. As the server must only be able to verify that a password is correct, there is no need to store the actual passwords themselves.

To strengthen security one needs to make it difficult for many passwords to be tried in a short time. This can be achieved by the following measures:

- Restrict access to hashed passwords.
- Enforce the use of longer and more complex passwords. The key space of 8-character alphanumeric password is 18 million times the size of 5-character lower-case passwords.
- Employ computationally more expensive hash algorithms like MD5 [36] or SHA-1

[10].

- Use salted hash functions: a random string is used as an additional parameter to the hash function. This reduces the probability of attacks with pre-build dictionaries. In such a scenario, the attacker builds a database of inputs and corresponding hash values. Now if a hash value becomes known, only a database lookup is needed to find the matching password. However, with every bit of additional input from the salt, the size of such a database would double.

4.3 Authorization

After authentication, the leaf GDS server knows the client's identity. Then it needs to decide if the client has the right to perform an operation on the directory. The server keeps an Access Control List (ACL), and it authorizes the client's request by comparing the client's identity with the corresponding record in the ACL.

4.4 Security Solutions

There are several ways to implement security in GDS. Because we are using CORBA and Java, either we could use an ORB that provides a security layer or we could develop user-application level security layer.

CORBA Security Service and Java Cryptography Architecture are discussed in the following sections.

4.4.1 CORBA Security Service

The CORBA Security Service is designed to provide a security architecture that can support a variety of security policies to meet different needs. It hides the security mechanism from both the servers and clients. It implements link security, and needs to encrypt the IIOP messages. Thus it has to be the first and last service to do so. The full specification of the Security Service can be found at [31]. The specification comprises:

- Identification and authentication.
- Authorization and infrastructure based access control.
- Security auditing.
- Security of communication.

* Non-repudiation.

We discussed about authentication and authorization above. Security auditing is to correctly identify the user, even after a chain of calls through many objects. To ensure secure communication, which is usually over insecure lower layer communications, requires trust to be established between the client and server, which may include authentication of both sides and may require integrity protection and confidentiality protection of messages in transit between them. Non-repudiation provides irrefutable evidence of actions such as proof of origin of data to the recipient, or proof of receipt of data to the sender to protect against subsequent attempts to falsely deny the receiving or sending of the data.

The CORBA Security Service can control access to an application object without it being aware of security, so it can be ported to environments that enforce different security policies and use different security mechanisms. However, if an object requires application level security, the security attributes must be delegated and made available to the application for access control. An example of CORBA Security Service is Jacorb[22].

4.4.2 Java Cryptography Architecture

Java Cryptography Architecture (JCA) [23], which is included in the java.security package, provides application level security. Three useful cryptographic concepts are useful for authentication:

- Message digests produce a small “fingerprint” of a larger set of data.
- Digital signatures can prove the integrity of data.
- Certificates are used as cryptographically safe containers for pubic keys.

4.4.3 GDS Security Mechanism

GDS uses authentication and authorization. At the beginning of a connection, the client’s identity is established during the authentication stage. For each subsequent operation requested by the client, the server checks whether the client is authorized to do so.

It uses salted message digest method from Java Cryptography Architecture to encrypt or decrypt the message at the two ends of client-server connection.

Message digests take an arbitrary amount of input data and computes it into a short digest. The algorithms for the computation are usually complicated. To avoid sending clear-text password, we can make a message digest of the password and send it to the server. The server uses the same algorithm on its copy of the client’s password and compares the two digests. If they are equal, then the client is authenticated. However, this method is venerable to the replay attack. A replay attack is when a malicious user first listened to the communication channel and captured the digested password. Then he reuses it to gain access to the server. To improve this mechanism, we can use a salted message digest. The “salt” usually contains session information. We add the “salt” to the password, and produce a message digest from the “salted” message. These “salt” must also be sent to the server, so that the server can use it to calculate a matching digest value. The server must be programmed to expect the extra information and include it in its message digest calculations.

The Java Cryptography Architecture (JCA) makes it very easy to use message digests. The java .security.MessageDigest class encapsulates a cryptographic message digest. In our implementation, we use a random number and a timestamp of the connection session as the “salt”, shown in Figure 4.1.

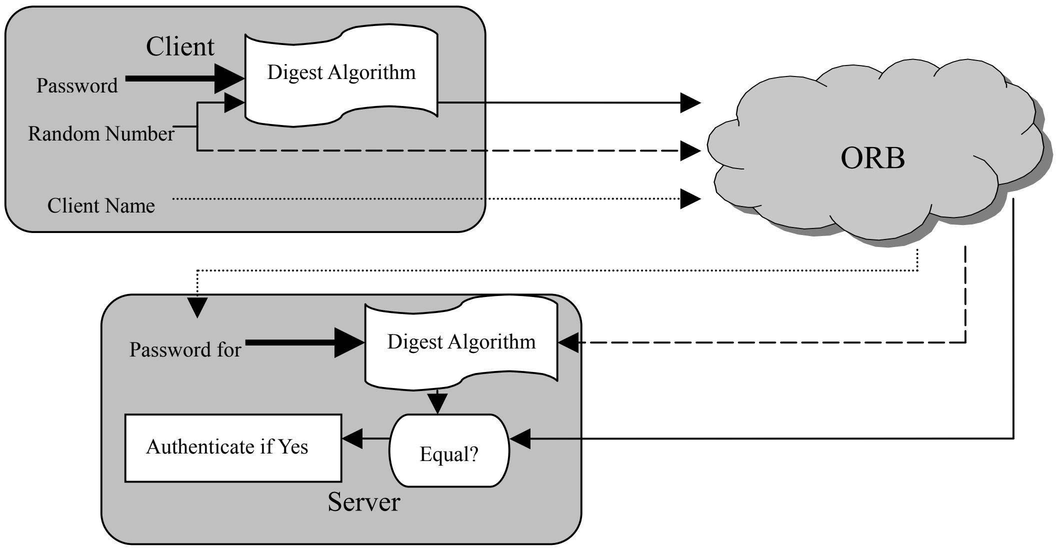


Figure 4.1: Message Digest

Chapter 5
Replication and Consistency

5.1 GDS Consistency Issues

In a leaf server, the shared directory entries are consistent because all the functions accessing them are synchronized. In the current design of GridStat, only the original server is used. Its replicas are only used for replication; they are not involved in the communication with other clients or servers. Thus processes will not update data entries simultaneously from different replicas.

The leaf GDS servers keep all the directory entries. Failure of a leaf server will cause loss of critical information and make the GDS inconsistent. The internal servers are links among different leaf servers. Failure of an internal server will break the connectivity of the GDS server hierarchy.

In GDS, when a server fails, its client or child server tries to reconnect to the sever a limited number of times at a certain rate. These parameters should be set depending on the real application and recovery time.

5.2 Replication

The replication of data is an important issue in GDS.

Informally, replication consists of maintaining identical copies of data and process in several different locations. There are two primary reasons for replicating data: reliability and performance, [4]. Replication is a technique to achieve fault tolerance, [32]. If one replica fails, others are still available to provide service. Replication can also be used to improve the performance of a system by providing closer access of data to the clients.

One of the major problems is to keep the replicas consistent. One design issue is whether data updates are pulled or pushed, [3].

5.2.1 Pull-based Protocols

In a pull-based or a client-based protocol, a replica requests the server to the send it all the updates since the last replication time. This is a common strategy applied to Web caches. This protocol is efficient when the read-to-update ratio is relatively low. There are a couple of proposed standard replication mechanisms for LDAP. They are all follow the pull-based approach.

In LCUP (LDAP Client Update Protocol), [35], the server implementing LCUP protocol provides a cookie to the replica containing the replica's state information. Each time when the replica replicates the server’s data, the server calls the replica and sends back this cookie and the server takes care of returning only the changes done on the server after the last replication from this replica.

Another way is suggested in [37]; use timestamps for replication. The replica stores the timestamp of the last update for the directory. Each entry in the directory has a timestamp. In each replication, the replica compares the last update timestamp with the directory entries’ timestamp on the server. If the timestamp of an entry is later then the timestamp the replica has, it is updated with the one on the server.

This methodology does not require the server to do anything special to support replication except to maintain the last update time stamp attribute for each entry, as well as allow access to this attribute so that the replica can compare its last update time stamp for the entry. On the other hand this does have an impact on performance since at the time of replication each entry’s timestamp needs to be compared to decide which entries in the client side needs to be updated.

5.2.2 Push-based Replication

In contrast, in a push-base approach, also referred to as server-based protocols, updates are propagated to all the replicas without their requesting. Push-based protocols are applied when replicas generally need to maintain a relatively high degree of consistency. This is usually used when the read-to-update ratio is relatively high.

5.2.3 GDS Replication Mechanism

In GDS, a replica server has all the critical data of the original server. For a leaf GDS server, the critical data is its name, the data storage (DS), the access control list (ACL) and the password file. For an internal GDS server, the critical data is its name, path, and children's names. After a GDS server is started, the replica server can be started. It first needs to be authenticated by the server. Then it can do replication.

We use both pull-based and push-based replication. Pull-based replication is used when read-to-update ratio is low while push-based replication is used when the read-to-update ratio is high. The choice of choosing a pull-based or a push-based mechanism is based on the goal of using fewer operations and making the directory more consistent.

In the pull-based way, we use a simplified replication mechanism that combines both LCUP and the timestamp. The algorithm is shown in figure 5.1. A server has a client log file for each replica, and the log file has a timestamp of last replication time from this replica. The server writes updates into each the replicas' log file. Each time the replica replicates the server, it compares the its current time with the timestamp in the log file and updates the if the current time is greater. Then it clears the log file, and marks it with the new timestamp.

```
Requirement: Replica log file (timestamped)

   1. Initial step:
         a. Replica calls the server
         b. duplicates the data files
         c. creates its log file
   2. Each time server updates data entries:
         a. it writes to all replica's log files
   3. Replica calls the server at a certain rate:
         a. get updates in its log
         b. clear it, put a new timestamp
```

Figure 5.1: Pull-based Replication Algorithm Outline

Push-based replication is logically simpler. Each time the server updates its directory, it pushes that update into all of its replicas. The algorithm is in figure 5.2.

```
Requirement: A handle for each replica

   1. Initial step:
         a. Replica connects to the server
         b. Duplicates the data files
   2. Each time server updates:
         a. It pushes the update to each replica
   3. Each time replica gets new update:
         a. It updates its own data files
```

Figure 5.2: Push-based Replication Algorithm Outline

In our current implementation, a replica is started manually. In a real application, the replicas might need to be server initialized. At any time, a server maintains a certain number of replicas and assigns their priorities. If one replica fails, it starts a new one.

Currently replication is only used to improve servers' availability. Placing replicas close to the clients can improve system's performance by reducing communication delay. This is discussed in the future work chapter.

5.3 Recovery

When a GDS server fails, all the operations the server is actively processing and all the communications with other servers and clients will fail. During the time in which a replica is replacing the failed server, all the messages sent to the failed server will be discarded.

The replica checks the server's existence at a certain rate. If it detects the failure, it will then start the recovery process. It first connects to the failed server's parent server, replaces the child object of the failed server with itself and invokes the parent to connect to it. It also registers itself with the same name in the CORBA naming service. After recovery, the new server starts to operate like the previous server. The clients and parent server can communicate with the new server in exactly the same way as before the server's failure. The algorithm is shown in figure 5.3.

```
1. A replica checks the server's existence at a certain rate
      a. For Pull-based replication, this is done in the same
         step with replication
2. If it detects server's failure, it
      a. changes into server mode
      b. registers itself with the same name as the sever
      c. connects to the server's parent
3. Clients or child servers reconnects
```

Figure 5.3: Recovery Algorithm Outline

5.4 Lease

If a publisher fails, it stops publishing all the status variables, however, their entries still remain in the directory without being deleted. Thus, the system becomes inconsistent.

We use a lease for each entry. After a status item is added to the directory by its publisher, it is issued a lease with a certain length of time period. Publisher must renew the leases for all of its status variables to continue their registration. If the lease is not renewed the entry will be automatically deleted.

Chapter 6
Implementation

6.1 CORBA IDL

We wrote two IDLs in GDS, Leaf.IDL (Figure 6.1) for leaf server and Internal.IDL (Figure 6.2) for internal server.

```
// The IDL file for leaf GDS server
module GDS {
        module Leaf {
          interface Leaf{
                string send(in string msg);
                string replicate(in string replicname);
          };
        };
```

Figure 6.1: IDL for Leaf GDS Server

The ‘send’ function can be called by two different types of entities. A client invokes the ‘send’ function with the string representation of an XML message. ‘send’ processes the request, and then returns the string representation of the response XML message back to the client. A parent server forwards a search request to leaf server, it also calls the ‘send’ function .

The ‘replicate’ function is called by a replica of the current server. The replica calls this function at a certain rate, this rate decided by the system policy and the recovery time.

Each time the replica duplicates the critical files from the original server. If the server fails, the replica will start recovery process.

```
// The IDL file for internal GDS server
module GDS {
        module Internal {
          interface Internal{
                string forward(in string msg);
                boolean bindChild(in string childname);
                string replicate(in string replicname);
          };
        };
};
```

Figure 6.2: IDL for Internal GDS Server

The 'forward' function forwards requests from child server to parent server or from parent server to child server. The message is using the simple server communication protocol we described in chapter 4.

When a child server comes into the GDS network, it calls the 'bindChild' function of its parent, and notifies the parent to set up a connection to it. Thus a parent and child are two-way connected.

6.2 Leaf GDS server

The leaf QoS broker has a clientList of bound clients, a password file for client authentication, an access control file for client authorization, a data storage place to store user’s directory entries and an object handle of its parent. See Figure 6.3.

6.2.1 Client List

Each leaf QoS broker has a client List. It is a list of connected clients. After a client is bound to the QoS broker, it is added to the client List and a connection session between the client and the leaf QoS broker starts. One client can only have one connection session with its leaf QoS broker. It will be removed from the client List after an Unbind operation or a connection session timeout. If another Bind request from the same client is received before session timeout, the length of the connection session will be extended. See Figure 6.2.1.

The length of the timeout for each session can be decided by the leaf QoS broker, the default value is set to be “DefaultTimeOut”. There is a thread wakes up at a certain interval and check if any client entry is timeout, if so, it’ll delete that entry.

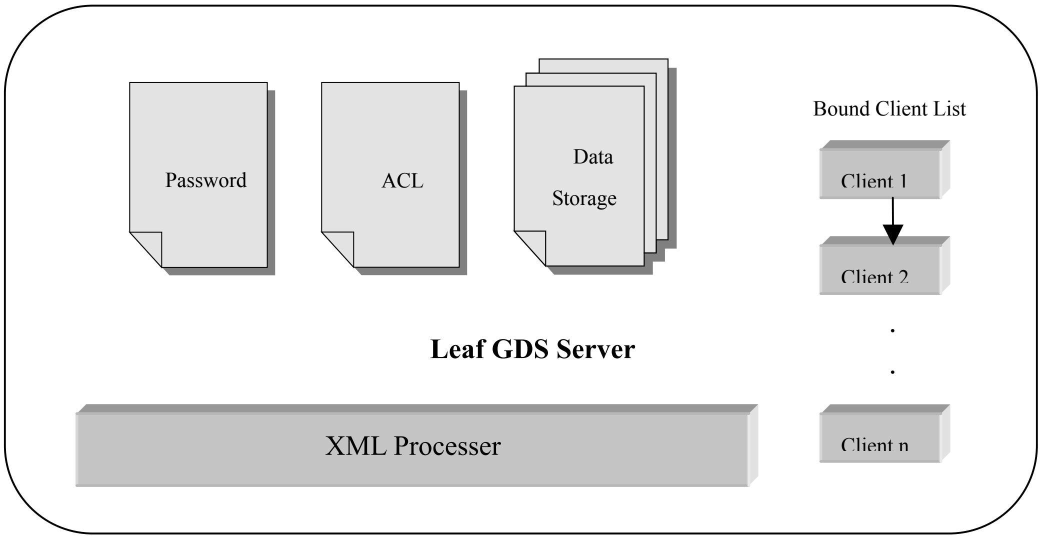


Figure 6.3: Leaf GDS server Structure

6.2.2 Data Storage

Each leaf QoS broker has a data storage place to store the data. It would be desirable to use database such as MySQL[29] as a repository. A database provides ACID model: atomicity, consistency, isolation and durability. In our current implementation, we temporarily use files with extension ".ds". Entries are sorted in the file.

The data storage, password file, and access control file are shared data. To maintain data consistency, mutual exclusion needs to be provided when accessing the data. In Java, this is easy because each object is a monitor. For example, a data storage file has add, delete, search, modify functions that can be called by different threads. To ensure safety simply adds a “Synchronized” tag in front of the function definition, then only one of them can be called at a time, and other function must wait until the current one finished.

6.3 Internal GDS Server

The data structure internal server is simpler than the leaf server is. It has the object references to its children and parent, a pool of threads to handle parallel computing for search and a semaphore to coordinate among threads.

6.4 Edge Status Routers

All the GDS servers are interconnected according to its topology during its existence. After creation, each leaf QoS broker tries to connect to its internal QoS broker, and the internal QoS broker tries to connect to its parent internal QoS broker until the root QoS broker is reached. So on and so forth, the QoS brokers’ tree topology is formed.

In the design of the GridStat network, a publisher or a subscriber can only talk with the leaf QoS broker (or leaf GDS server) via the Edge Status Router. This is because the goal for GridStat is to make the communication fast and reliable. Once it finds the shortest path between the publisher and the subscriber, it would not interfere with the QoS broker’s network. Thus it is very import to make the Status Routers as simple as possible. In the

GDS itself, the Edge Status Router is not needed. But it is only way we can talk to the GDS servers. They need to forward the request from the clients to the leaf QoS broker. The Edge Status Routers gets information from a client, puts it into a XML formated message, and then submits it to its leaf QoS broker. It then gets the XML response from the leaf QoS broker, and then sends the result back to the client.

6.5 GDS Client

Due to that the communication between the publisher/Subscriber and the Edge Status Router is simple, in our implementation we made a GUI to talk to the Edge Status Routers as a client.

6.6 Other Implementation Issue

6.6.1 XML and Java Conversion

XML Schema was originally proposed by Microsoft, but became and official W3C recommendation in May 2001. The reason why we are using XML schema is because XML schemas support namespaces. It is written in XML and the tags used to define data are richer and more useful and it is extensible to future additions.

A typical XML element and complex type definition is like this:

```
<!-- Complex Type Definition: OptionalAttributes -->
<xs:complexType name="OptionalAttributes">
      <xs:sequence>
      <xs:element name="attributeName" type="xs:string" />
      <xs:element name="attributeType">
         <xs:simpleType>
               <xs:restriction base="xs:string">
                  <xs:enumeration value="Integer" />
                  <xs:enumeration value="Float" />
                  <xs:enumeration value="String" />
               </xs:restriction>
            </xs:simpleType>
      </xs:element>
      <xs:element name="attributeValue" type="xs:string" />
      </xs:sequence>
   </xs:complexType>
```

Figure 6.4: XML Schema for Type OptionalAttributes

This is a new complex type definition; its name is "OptionalAttributes". Inside <xs:sequence> and </xs:sequence> bracket, everything in the target xml file must be in the exact order. The "OptionalAttributes" has two string elements "attributeName" and "attributeValue", a restricted string element "attributeType" with only options of "Integer", "Float" and "String".

```
<!-- Add -->
  <xs:element name="addRequest">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="ClientName" type="xs:string" />
        <xs:element name="PublishItemName" type="xs:string" />
        <xs:element name="PublishRate" type="xs:positiveInteger" />
        <xs:element name="OptionalAttributeList"
                     type="OptionalAttributes"
                     minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

Figure 6.5: XML Schema for Add Request

This is the definition of the element named “addRequest”. It is a complexType element. It has a sequence of elements “ClientName”, “PublishItemName”, “PublisheRate” and “OptionalAttributeList”. The “OptionalAttributeList” has the type “OpionalAttributes” that we just defined, it may contain an unlimited number of user-defined attributes.

There are many tools available to provide packages for conversions between XML Schema and Java Classes, and between Java Objects and XML files.

After a XML Schema is written according to the specification in Chapter 3, we use Castor Source Generator to generate Java Classes from it. Castor Source Generator treats the XML Schema structures such as <complexType> and element in two main ways. The “element” method creates classes for all elements whose type is a <complexType>.

Abstract classes are created for all top-level <complexTypes>. Any elements whose type is a top-level type will have a new class created that extends the abstract class of the <complexType>. Classes are not created for elements whose type is <simpleType>. All the <simpleType> definitions will be created into a package named “types”.

We will not analyze the Java objects here, because they are generated binary files, and not human readable.

Most of the tools provide three basic functions. Marshall() takes a Java object, and converts it into an XML file. Unmarshall() converts an XML file back to the Java object. Each created java class has a Validate() function that can validate if an instance of the class has conflict with the XML schema.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Message>
    <MessageID>56</MessageID>
    <MessageType>addRequest</MessageType>
    <addRequest>
        <ClientName>Pullman Sub Station</ClientName>
        <PublishItem>Voltage</PublishItem>
        <PublishRate>20</PublishRate>
        <OptionalAttributeList>
            <attributeName>Location</attributeName>
            <attributeType>String</attributeType>
            <attributeValue>Pullman</attributeValue>
        </OptionalAttributeList>
    </addRequest>
  </Message>
```

Figure 6.6: Sample XML representation of Add Request

This is an xml file created based on the XML Schema piece before.

Though most of the source generators perform similar functionalities, there are some minor differences. Thus when you write the XML Schema, you should refer to the restrictions of the source generator you chose.

6.6.2 Search Algorithm

A good search algorithm should utilize only the GDS resources needed. A bad design would either be search all the leaf GDS servers at the same time or search one at a time.

The number of servers that a search should start with depends on the maxEntryReturn attribute in the request. Each internal GDS server keeps track of the child leaf server quantity. After receiving the search or find path request from the client, the leaf GDS server first searches its own directory. If the number of results is less than the number of return entries requested, it then forwards the request to its parent.

Leaf GDS server:

```
ResultType Search(SearchCriteria, maxEntryReturn) {
   Result = Search(SearchCriteria);
   if (Result not empty)
   maxEntryReturn -= Number of results found
   if (maxEntryReturn > 0) {
         msg = SearchCriteria + callerName + maxEntryReturn
         parentResult = ParentServer.forward(msg);
         Result = parentResult + Result
   }
   return Result;
 }
```

Figure 6.7: Search Algorithm in Leaf GDS Server

Upon receiving the requests form its children, the parent Server asks all of its child servers except the caller to search their directories or their children's directories. The search is finished and returned when the search request is satisfied or when it searches the whole directory.

Internal GDS server:

```
String forward (msg) {
    Criteria = msg.getCriteria
    remainEntry = msg.getMaxEntry
    callerName = msg.getCallerName
    if (children are leaf Servers)
    Threads = new CallLeafServerThread[ChildNum];
    else (children are internal Servers)
        Threads = new CallInternalServerThread[ChildNum];

    // start thread per child, except the caller child if any
    for(i: 0 to ChildNum) {
        if(Child[i].Name != callerName) {
            pass (ChildLeafObject[i], SearchCriteria) to Threads[i]
            Threads[i].start();
        }
    }

    //wait until all the threads are done

    for(i: 0 to ChildNum) {
        if(Child[i].Name != callerName) {
            Result += Threads[i].Result
            remainEntry -= Number of results found in server[i]
        }
    }

    //if still need to forward up
    if(maxEntry > 0) {
        if(Parent != null) { //if current server is not the root
            Result = Result + ParentServer.forward(msg);
        }
    }
    return Result;
  }
```

Figure 6.8: Search Algorithm in Internal GDS Server

6.6.3 Interface to GridStat and client Development

To add all the GDS functionalities to GridStat, the new functions in IDL files have to be added to the GridStat server's IDLs. And all of the GDS package need to be imported into the GridStat project.

It is easy for the clients to develop their own software to talk to GDS. There are two ways:

In our implementation, we develop functions "Bind(), Unbind(), Add(), Delete(), Modify(), FindPath() and Search()". Each function sends a request and waits for the response. The user can call those functions directly. All the functions accept common data, they hide the conversion between Java and XML.

Based on the xml schema, the user can make their xml messages and invoke the ping() function in the Edge Status Router and talk with the GDS servers.

Chapter 7
Performance Analysis

7.1 Test environment

All the tests are performed on a computer with an Intel Pentium 4 2.40GHz CPU, with 512M RAM and Windows XP Professional.

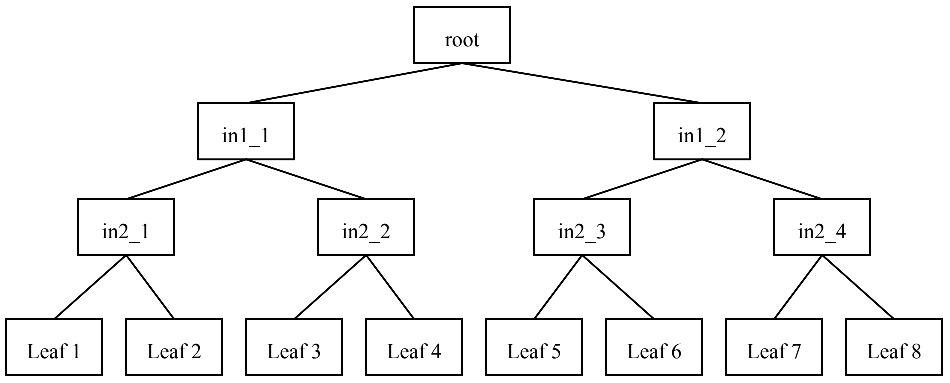
7.2 Test setting

We separate GDS operations into four categories:

- Client-leaf GDS server communication (bind): Bind.
- Client-leaf GDS server communication (unbind): Unbind.
- Client-leaf GDS server communication (data access): Add, Delete, Modify.
- Client-GDS servers communication: FindPath, Search.

In each category, because the communication and the overall algorithms are similar among the functions, their performances are similar. In client-leaf server communication level, Bind performs authentication; Unbind only needs authorization, it is the only operation that does not send a response back to the client; Add, Delete, Modify operations first have to be authorized, then they will perform directory data access. Finally, FindPath and Search involves the communication between client and multiple servers for search continuation.

We created four different networks to test GDS. Test network one, two, three and four (TN1, TN2, TN3, TN4) are full binary trees with height one through four. The structure of TN3 is in Figure 7.2.



In each server, only relative distinguish name (RDN) is listed.

Figure 7.1: Test Network Structure 3 (TN3)

Binary tree's parent-child dependency provides a simple example of GDS hierarchy for testing the performance of the implemenation. The parameters in each test network are shown in Table 1.

Name	TN1	TN2	TN3	TN4
Internal servers	1	3	7	15
Leaf servers	2	4	8	16
Load time (ms)	878	3621	7352	15169

Table 7.1: Test Network Parameters

7.3 Test result and analysis

7.3.1 The delay time of Bind, Unbind and Add

We test the performance of the bind, add and unbind operation by doing a bind on one client, adding an item and then unbinding it. The reason we compare their delay time is because all of these functions only involve the communication between a client and its leaf server. A bind operation encrypts a client's credential in the bind request, converts the bind request into XML format and sends it to the server, the server authenticates the client by decrypting and verifying the credential, then send a response back. An Add operation sends a XML format request, and waits for the response. Meanwhile, after the server receives the request, it processes it, updates the directory entries and sends a response back. An Unbind operation sends an unbind request to server, the server then deletes the client from the client list. The result of bind and unbind in different test networks is shown in Figure 7.2.

The test of these operations is based on the length of a single operation, not an average of a set of operations.

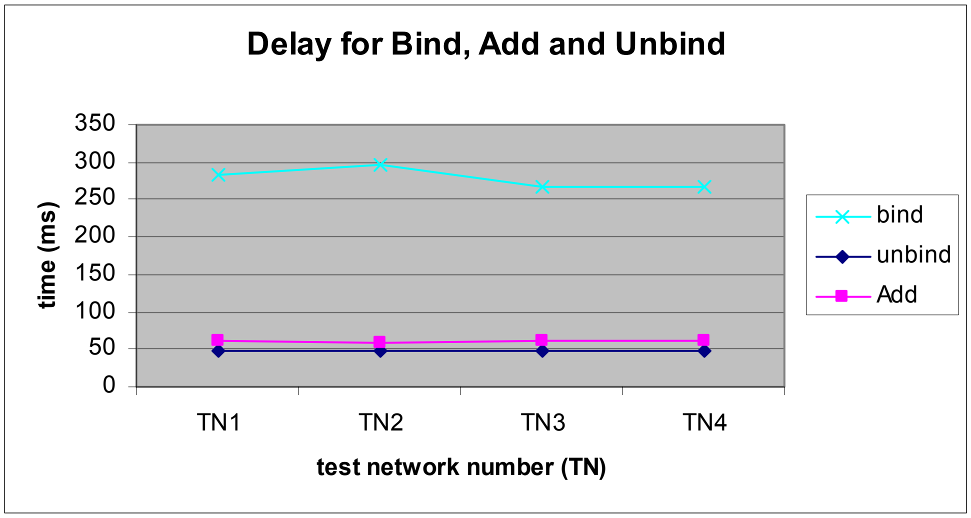


Figure 7.2: Delay Time of Bind, Add and Unbind Operations

The time of Bind, Add and Unbind operations in different test networks are very similar. This is because they are client-leaf GDS server type communication. No matter how big the network is, these operations only involve the communication between the client and its leaf server. This is also the case in Add, Delete and Modify operations.

A Bind operation takes about 270ms, it spends a certain amount of its time encrypting and decrypting the message. Besides, it initiates a connection to the server. Without these steps, an Add operation that takes about 60ms, is much faster. They are both slower than an Unbind operation which does not even need a response. An Unbind operation takes about 47ms.

7.3.2 Add

In this test, we let one client randomly add 10, 20, …, 100 entries repeatedly to its leaf GDS server. Because the communication with the leaf server is local so the time doesn’t depend on the size of the test network. This test is done in test network TN1.

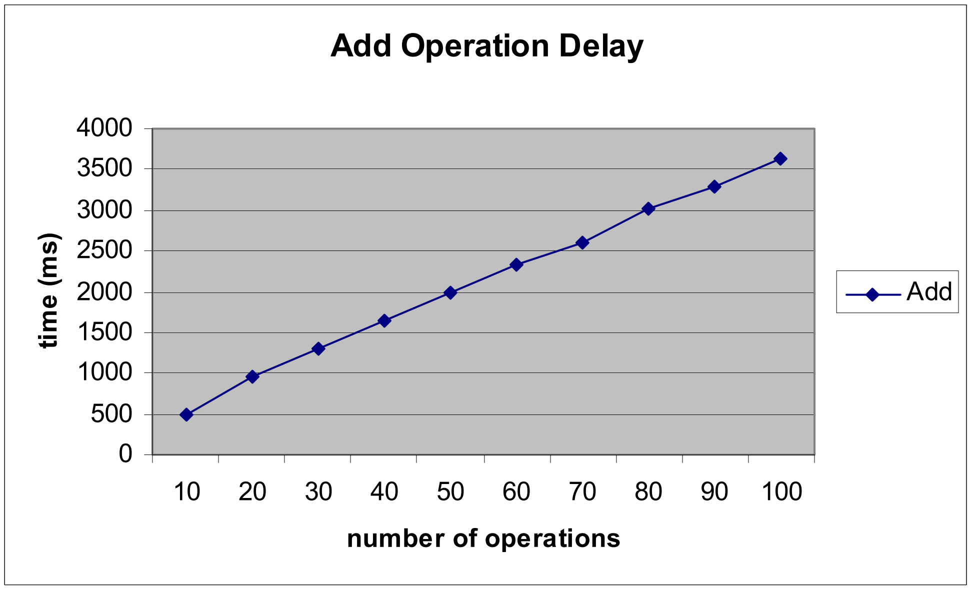


Figure 7.3: Delay Time of Add Operations

Because the add operations do not affect each other, the growth in the Figure 7.3 is linear. On average, an add operation takes 40ms.

7.3.3 FindPath Single Operation

In Test 3, we want to test the time range of a FindPath operation. It takes the smallest amount of time when FindPath finds an entry in the first leaf server, and it takes the largest amount of time when it does not find the entry. In the latter case, FindPath traverse the whole the directory.

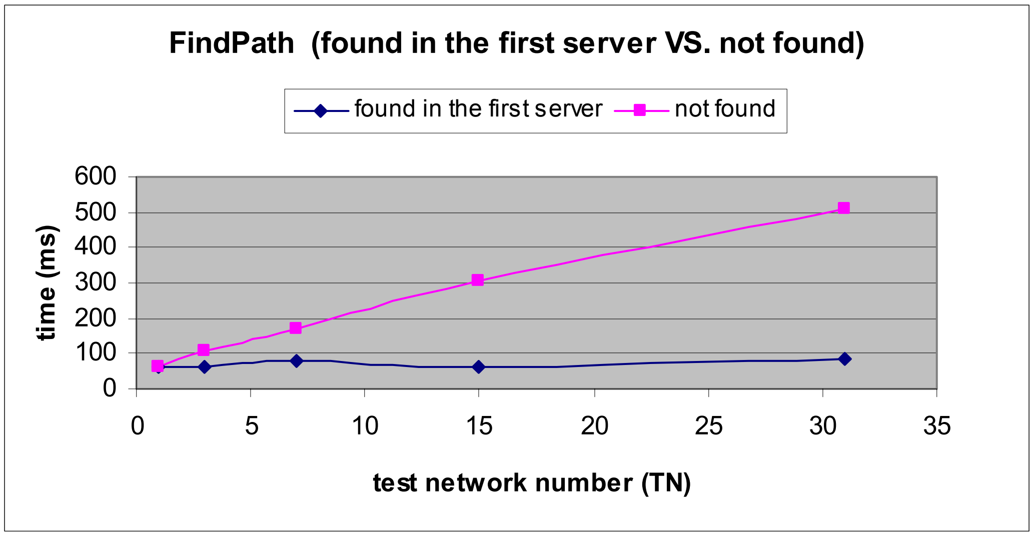


Figure 7.4: Delay Time of a Success and Unsuccessful FindPath Operation

When FindPath finds the entry in the first leaf server it visits, it takes a similar amount of time among all the test networks. Because at this time, it is a client-leaf server type communication. When it is not found, it will leverage every server in the GDS. This gives us an idea of how long it takes to traverse the corresponding test network.

7.3.4 FindPath

Call FindPath 5, 10, 50, 100, 500 and 1000 times to search a random entry with maximum number of return entries 1, and set the timeout to a large number (large enough to let each search finish). Before we start this test, we randomly add 100 entries in each leaf server. Thus, there are a total of 1,600 entries in the directory.

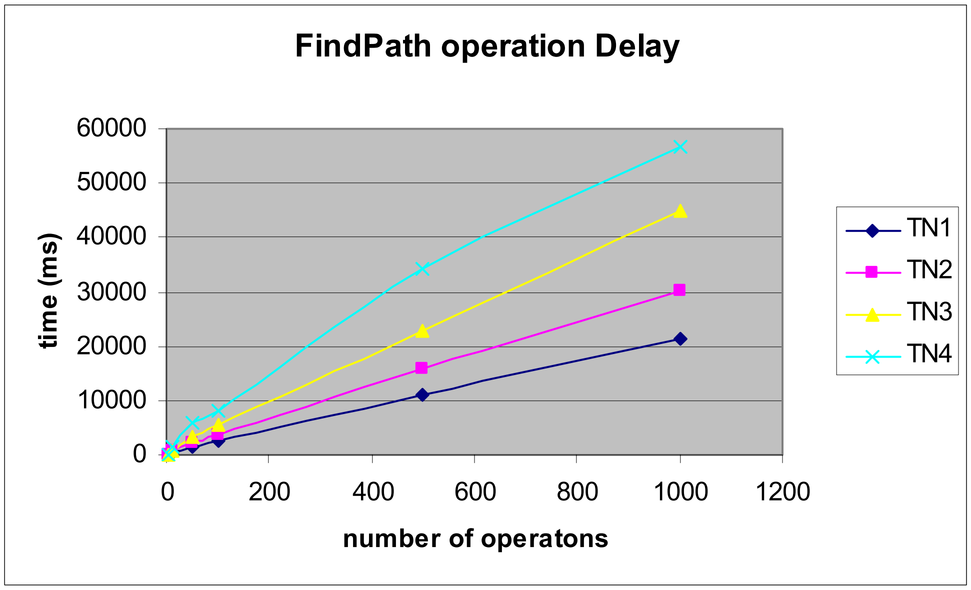


Figure 7.5: Delay time of FindPath Operations

One observation is that, the time of the FindPath operation grows slower than the number of servers and the number of directory entries in the GDS. This is because in the same level of the servers tree, searching is done in parallel. Thus, the time of the FindPath operation is a linear increase with the height of the test network.

7.3.5 Leaf Server Pull-based Replication & Recovery

We test the delay time of leaf server replication and recovery. In this test, 'First time replicate' measures the time of a replica first bind to a server and replicate the data files.

Leaf GDS server	First time replicate	Replicate (no update)	Recovery
Time (ms)	234	0 ~ 15	62

Table 7.2: Leaf GDS Server Pull-based Replication and Recovery Time

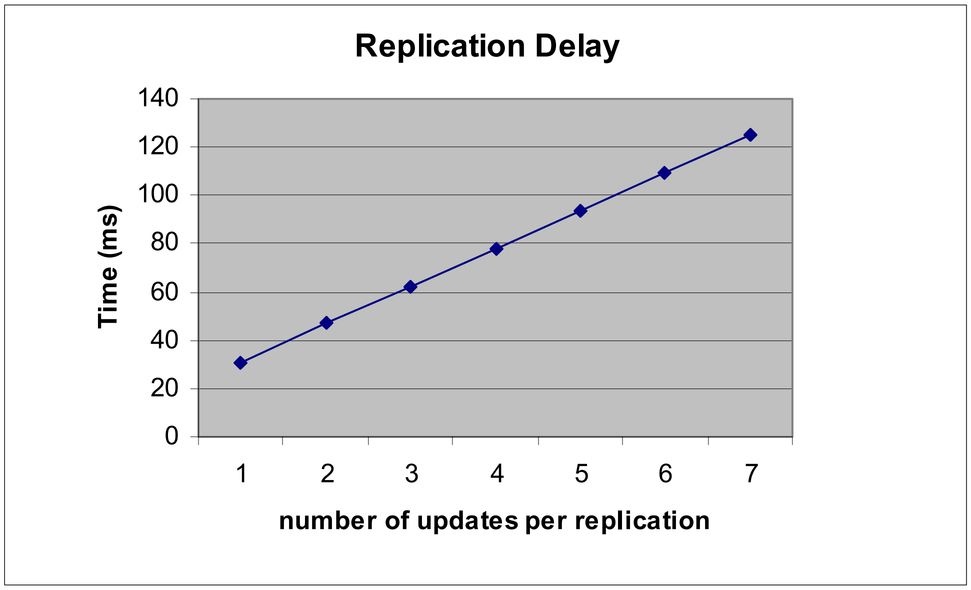


Figure 7.6: Pull-based Replication Delay

The first time it takes about 234 ms for a replica to replicate the leaf server, because it includes finding the server in the naming service and setting up the connection between the server, authentication and the replication.

After that it takes about 0~15 ms each time for taking to the server when there is no updates. Comparing to 47 ms for an Unbind operation, the transmission between server and client is much faster without XML conversion. The delay time of different number of updates per replication is shown in figure 7.6. All the updates are sent in one message, and for each update, the replica needs to update its own directory accordingly. The growth is linear.

It takes about 62 ms for the replica to replace the server. However, because it checks the server at a certain interval, the actual recovery time needs to add the time of check interval.

7.3.6 Leaf Server Push-based Replication and Recovery

Internal GDS server	First time replicate	Replicate	Recovery
Time (ms)	234	47 ~ 78	63

Table 7.3: Leaf GDS Server Push-based Replication and Recovery Time

Similarly to the leaf server, the first time for a replication of an internal server is longer. It takes about 234 ms. Each time, the server pushes an update to its replicas, it takes about 47~78ms. This is done in parallel with server's other operations.

However, our test is run on a single machine, the time of the remote function call in a large connected distributed system takes much more time.

It takes a similar amount of time for a push-based replica to recover, but this is not finished development. The recovery time in Table 2 and Table 3 only includes the time for the replica to replace the failed internal server in the naming service and the parent server.

It does not include the time that all of the server’s children take to reconnect to it. Because a child server tries to reconnect to its parent server after communication failure at a certain rate and it will only try a certain number of times. These times need to add into the total recovery time of an internal server.

Chapter 8
Conclusion and Future Work

8.1 Conclusion

GDS is a complete directory service. It contains the 5 elements defined by X.500 for a directory service [17]:

A directory service has a model that defines the basic structure in which information is stored. In GDS, each QoS broker maintains data storage for directory entries. Each entry is an object that contains a name, a path, a published item with its rate and a sequence of optional user defined attributes. The optional attribute consists of a name, type and value.

A directory service has a tree-like hierarchy. The tree-like structure of the GDS servers provides the distributed information management and lookup service.

A directory service has a collection of command functions that clients can use to manipulate directory entries. According to the requirement of the directory service for GridStat, these functions include Bind, Unbind, Add, Delete, Modify, FindPath and Search.

A directory service has a system to authenticate users. As a management and security concern, the directory service must authenticate users before granting them access to the directory. GDS supports different level of authentication. Each operation performed by a client needs to be authorized. GDS security is designed to thwart malicious modification to directory entries. In GridStat, clients are still required to be authenticated before accessing data from publishers.

GDS has a distribution model that lets clients view the data stored on multiple servers as a single entity. If a server (QoS manager in GridStat) receives a request for the data not in its domain, it passes the request to the other servers.

To provide a good directory service for GridStat network, many things come into consideration: the scalability of GridStat, the delay requirement. GridStat network needs to be able to scale to all the interesting sensor units. The sensor unit is a measurement system monitoring a certain area. Even at the granularity of cities, the number of directory entries used in GDS will be on the order of 20,000. The directory entries need to be distributed into different servers. Duplicating same data into all the servers raises the problem of consistency and increases the communication delay.

Since GDS does not deal with real-time data, communication delay is not extremely critical. As we described in section 7.3.5, using XML makes the communication slower. But in the sense of providing a standardized user interface, and making GDS more easily ported into a web services provider, XML plays an important role.

The decision between using pull-based replication and push-based replication depends on the read-to-update rate. The goal is to maintain the server-replicas' data synchronization and reduce the number of replication operations. In the current implementation of GDS, this is a configuration choice. When the rate is high, push-based replication performs better, whereas when the rate is low, pull-based replication performs better.

8.2 Future work

To improve the search performance, a search cache can be used.

A real database should replace the current data storage file. This will increase scalability and lower file operation times. Also, a database indexes the entries, which can save database search time.

The GDS should have more security considerations. For example, public key infrastructure (PKI) [38] should be used for authentication.

Using replicas to improve performance is not discussed in this thesis, because currently it is not designed in the GridStat hierarchy. But it might be future improvement of GridStat.

Appendix A: XML Schema

```
<?xml version="1.0" ?>
<xs:schema xmlns:xs=http://www.w3.org/2001/XMLSchema
                              xmlns="www.eecs.wsu.edu">

<!-- simple types definition -->
<!-- Standard Response -->
<xs:element name="stdResponse">
      <xs:simpleType>
            <xs:restriction base="xs:string">
                  <xs:enumeration value="Success" />
                  <xs:enumeration value="MsgInvalid" />
                  <xs:enumeration value="NotBinded" />
                  <xs:enumeration value="AccessDenied" />
                  <xs:enumeration value="UnknownError" />
            </xs:restriction>
      </xs:simpleType>
</xs:element>

<!-- complex types definition -->
<!-- complex type: OptionalAttributes -->
<xs:complexType name="OptionalAttributes">
      <xs:sequence>
            <xs:element name="attributeName" type="xs:string" />
            <xs:element name="attributeType">
                  <xs:simpleType>
                        <xs:restriction base="xs:string">
                              <xs:enumeration value="Integer" />
                              <xs:enumeration value="Float" />
                              <xs:enumeration value="String" />
                        </xs:restriction>
                  </xs:simpleType>
            </xs:element>
            <xs:element name="attributeValue" type="xs:string" />
```

```
        </xs:sequence>
</xs:complexType>

<!-- Complex Type: ModifyOptionalAttr -->
<xs:complexType name="ModifyOptionalAttr">
        <xs:sequence>
                <xs:element name="Operation">
                        <xs:simpleType>
                                <xs:restriction base="xs:string">
                                        <xs:enumeration value="Delete" />
                                        <xs:enumeration value="Replace" />
                                </xs:restriction>
                        </xs:simpleType>
                </xs:element>
                <xs:element name="optionalAttrbutes" type="OptionalAttributes" />
        </xs:sequence>
</xs:complexType>

<!-- complex elements definition -->
<!-- Message -->
<xs:element name="Message">
        <xs:complexType>
                <xs:sequence>
                <!-- Unique Message ID -->
                        <xs:element name="MessageID" type="xs:long" />
                        <xs:element name="MessageType" type="xs:string" />
                                <xs:choice>
                                        <xs:element ref="bindRequest" />
                                        <xs:element ref="bindResponse" />
                                        <xs:element ref="unbindRequest" />
                                        <xs:element ref="addRequest" />
                                        <xs:elemten ref="addResponse" />
                                        <xs:element ref="deleteRequest" />
                                        <xs:element ref="deleteResponse" />
                                        <xs:element ref="modifyRequest" />
                                        <xs:element ref="modifyResponse" />
                                        <xs:element ref="pathRequest" />
```

```
                    <xs:element ref="pathResponse" />
                    <xs:element ref="searchRequest" />
                    <xs:element ref="searchResponse" />
                </xs:choice>
        </xs:sequence>
    </xs:complexType>
</xs:element>

<!-- Bind -->
<!-- Bind request -->
<xs:element name="bindRequest">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="ClientName" type="xs:string" />
            <xs:element name="AuthMethod" type="xs:integer"/>
            <xs:element name="Credential" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>

<!-- Bind response-->
<xs:element name="bindResponse">
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:enumeration value="Success"/>
            <xs:enumeration value="MessageInvalid"/>
            <xs:enumeration value="MethodNotSupported"/>
            <xs:enumeration value="StrongAuthRequired"/>
            <xs:enumeration value="WrongCredentials"/>
            <xs:enumeration value="ServerUnavailable"/>
        </xs:restriction>
    </xs:simpleType>
</xs:element>

<!-- Unbind -->
<xs:element name="unbindRequest" type="xs:string" default="null"/>
```

```
<!-- Add -->
<!-- Add request -->
<xs:element name="addRequest">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="ClientName" type="xs:string" />
                        <xs:element name="PublishItemName" type="xs:string" />
                        <xs:element name="PublishRate" type="xs:positiveInteger" />
                        <xs:element  name="OptionalAttributeList"  type="OptionalAttributes"  minOccurs="0"
                        maxOccurs="unbounded" />
                </xs:sequence>
        </xs:complexType>
</xs:element>

<!-- Add response -->
<xs:element name="addResponse">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="stdResponse" minOccurs="0" />
                        <xs:element name="response" minOccurs="0" >
                                <xs:simpleType>
                                        <xs:restriction base="xs:string" >
                                                <xs:enumeration value="EntryExist" />
                                                <xs:enumeration value="DataStorageFailure" />
                                        </xs:restriction>
                                </xs:simpleType>
                        </xs:element>
                </xs:sequence>
        </xs:complexType>
</xs:element>

<!-- Delete -->
<xs:element name="deleteRequest">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="ClientName" type="xs:string"/>
                        <!--if Publisher is null, delete the everything for the client-->
```

```
            <xs:element name="PublisherItemName" type="xs:string" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>

<!-- Delete response -->
<xs:element name="addResponse">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="stdResponse" minOccurs="0" />
            <xs:element name="response" minOccurs="0" >
                <xs:simpleType>
                    <xs:restriction base="xs:string" >
                        <xs:enumeration value="EntryNotExist" />
                        <xs:enumeration value="DataStorageFailure" />
                    </xs:restriction>
                </xs:simpleType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
</xs:element>

<!-- Modify -->
<xs:element name="modifyRequest">
    <xs:complexType>
        <xs:all>
            <xs:element name="ClientName" type="xs:string" minOccurs="0" />
            <xs:element name="PublishItemName" type="xs:string" minOccurs="0" />
            <xs:element name="PublishRate" type="xs:positiveInteger" minOccurs="0" />
            <xs:element      name="modifyOptionalAttrList"      type="ModifyOptionalAttr"
            minOccurs="0" maxOccurs="unbounded" />
        </xs:all>
    </xs:complexType>
</xs:element>

<!-- Modify response -->
<xs:element name="addResponse">
```

```
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="stdResponse" minOccurs="0" />
                        <xs:element name="response" minOccurs="0" >
                                <xs:simpleType>
                                        <xs:restriction base="xs:string" >
                                                <xs:enumeration value="EntryNotExist" />
                                                <xs:enumeration value="DataStorageFailure" />
                                        </xs:restriction>
                                </xs:simpleType>
                        </xs:element>
                </xs:sequence>
        </xs:complexType>
</xs:element>

<!-- Request to Get Possible Paths for a Client Name-->
<xs:element name="pathRequest">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="ClientName" type="xs:string"/>
                        <xs:element name="Publisher" type="xs:string"/>
                        <xs:element name="maxEntryReturn" type="xs:positiveInteger" default="3"/>
                        <xs:element name="timeOut" type="xs:positiveInteger" default="10000" />
                </xs:sequence>
        </xs:complexType>
</xs:element>

<!-- Response for Get Paths-->
<xs:element name="pathResponse">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="stdResponse" minOccurs="0"/>
                        <xs:element name="Publisher" type="xs:string"/>
                        <xs:element name="ClientPath" type="xs:string" minOccurs="0" maxOccurs="10"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
```

```
<!-- Search -->
<xs:element name="searchRequest">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="ClientName" type="xs:string" />
                        <xs:element         name="Criteria"         type="OptionalAttributes"         minOccurs="0"
                        maxOccurs="unbounded" />
                        <xs:element name="Filter" minOccurs="0" >
                                <xs:simpleType>
                                        <xs:restriction base="xs:string">
                                                <xs:enumeration value="AND"/>
                                                <xs:enumeration value="OR"/>
                                                <xs:enumeration value="NOT"/>
                                                <xs:enumeration value="EQUAL"/>
                                                <xs:enumeration value="GE"/>
                                                <xs:enumeration value="LE"/>
                                                <xs:enumeration value="SUBSTRING"/>
                                        </xs:restriction>
                                </xs:simpleType>
                        </xs:element>
                        <xs:element name="maxEntryReturn" type="xs:positiveInteger" default="3"/>
                        <xs:element name="timeOut" type="xs:positiveInteger" default="10000" />
                </xs:sequence>
        </xs:complexType>
</xs:element>

<xs:element name="searchResponse">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="stdResponse" />
                        <xs:element name="ClientPath" type="xs:string" minOccurs="0" maxOccurs="10"/>
                <xs:sequence>
        </xs:complexType>
</xs:element>

</xs:schema>
```

Bibliography

[1] A. D. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder, "Grapevine: An exercise in distributed computing", Communication of ACM, 25(4): 260--274, April 1982.

[2] A. Freier, P. Karlton, and P. Kocher. "The SSL Protocol Version 3.0", Nov 1996.

[3] A. Tanenbaum and M. Steen, "Distributed Systems Principals and Paradigms", P291-360, Prentice Hall, 2002

[4] B. Press, "Data Structures and Algorithms with Object-Oriented Design Patterns in C++", p255, Wiley Text Books, Augest 1998.

[5] B. Venners, "Finding Services with the Jini Lookup Service – The Power and Limitations of the ServiceRegistrar Interface", JavaWorld, February 2000.

[6] "Breeze XML Binder", http://www.breezefactor.com/

[7] C. Weider, J. Fullton, and S. Spero, "Architecture of Whois++ Index Service", RFC1913, February 1996.

[8] C. Newman, "Using TLS with IMAP, POP3 and ACAP", RFC 2595, June 1999.

[9] "Castor Version 0.9.5.3", http://www.castor.org, March 2004.

[10] D. Eastlake, 3rd and P. Jones, "US Secure Hash Algorithm (SHA1)", RFC 3174, September 2001.

[11] D. Eastlake, IBM, "Domain Name System Security Extensions", RFC 2535, March

1999.

[12] “DNS Resource Directory”, http://www.dns.net/dnsrd/.

[13] “eXtensible Directory Access Protocol”, Network Working Group, Internet Draft, July 2001

[14] E. Ort and B. Mehta, “Java Architecture for XML Binding (JAXB)”, March 2003

[15] F. Yergeau, T. Bray, J. Paoli, C. Sperberg-McQueen, E. Maler, “Extensible Makeup Language (XML) 1.0 (Third Edition)”, www.w3.org/XML/, W3C Recommendation, February 2004.

[16] “Homepage of the Linux NIS/NIS+ Project”, www.linux-nis.org.

[17] Information technology, Open Systems Interconnection, “The Directory: Overview of concepts, models and service. Recommendation X.500”, International Telecommunications Union, Geneva, 1993.

[18] J. Gargano, K. Weiss, University of California, Davis, “Whois and Network Information Lookup Service, Whois++”, RFC 1834, August 1995.

[19] J. Hodges and R. Morgan, “Lightweight Directory Access Protocol (v3): Technical Specification “, Internet Draft, January 2001.

[20] J. Hodges, R. Morgan, and M. Wahl, “Lightweight Directory Access Protocol (v3): Extension for Transport Layer Security”, RFC 2830, IETF, May 2000.

[21] J. Snyder, “Distributed LDAP Features”, Network World, May 2000.

[22] “JacORB”, www.jacorb.org

[23] "Java Cryptography Architecture API Specification and Reference", http://java.sun.com/j2se/1.3/docs/guide/security/CryptoSpec.html, December 1999.

[24] K. Arnold, "The Jini Architecture: Dynamic Services in a Flexible Network", Sun Microsystems, Inc, Burlington, MA.

[25] K. Gjermundrod, I. Dionysion, D. Bakken, C. Hauer and A. Bosse, "Flexible and Robust Status Dissemination Middleware for the Electric Power Grid", Submitted for Journal Publication, September 2003.

[26] K. Harrenstien, M. Stahl, E. Feinler, "NICNAME/WHOIS", RFC 954, October 1985.

[27] K. Tomsovic, D. Bakken, V. Venkatasubramanian, A. Bose, "Designing the next generation of real-time control, communication and computations for large power systems", Submitted to IEEE Proceedings – Special Issue on Energy Infrastructure Systems, October 25, 2003.

[28] M. Hedin, "Comparing Java Data Binding Tools", http://www.xml.com/pub/a/2003/09/03/binding.html, September 2003.

[29] "MySQL", www.mysql.com

[30] Object Management Group (OMG), "Common Object Request Broker Architecture", www.corba.org

[31] Object Management Group (OMG), "Security Service, V 1.8" , www.omg.org/technology/documents/formal/security_service

[32] P. Verissimo, L. Rodrigues, "Distributed Systems for System Architects", P78, Kluwer Academic Publishers, Boston/Dordrecht/London, 2001.

[33] R. Bush, A. Durand, B. Fink, O. Gudmundsson, T. Hain, “Representing Internet Protocol version 6 (IPv6) Addresses in the Domain Name System (DNS), RFC 3363, August 2002.

[34] R. Dayal, “LDAP Replication Draft Analysis and Design Document”, www.mozilla.org/mailnews/arch/LDAP_replication2.html, February 2002.

[35] R. Megginson, M. Smith, O. Natkovich, J. Parham, “LDAP Client Update Protocol”, Internet Draft, August 2003.

[36] R. Rivest, “The MD5 Message-Digest Algorithm”, RFC 1321, IETF, April 1992.

[37] R. Weltman, M. Smith, M. Wahl, “LDAP Authorization Identity Request and Response Controls”, Internet-Draft, April 2003.

[38] S. Lloyd and C. Adams, “Understanding the Public-Key Infrastructure: Concepts, Standards, and Deployment Considerations”, P9-199, Que, November 1999.

[39] S. Williamson, M. Kosters, D. Blacka, J. Singh, K. Zeilstra, “RFC 2167Referral Whois (Rwhois) Protocol V1.5”, RFC 2167, Network Solutions, Inc. June, 1997.

[40] P. Windley, “Enabling Web Services”, www.windley.com, 2003

[41] Sun Microsystems, “The Java Web Services Tutorial”, www.java.sun.com/webservices/docs/1.3/tutorial, December 2003.

[42] “XGen”, http://www.commerceone.com/developers/docsoapxdk

[43] “XML Schema”, http://www.w3.org/2001/XMLSchema, March 2004.