# A Scalable Autonomous File-based Replica Management Framework

Dong Li

Master of Science Thesis
Stockholm, Sweden 2006
ICT/ECS-2006-50

# A Scalable Autonomous File-based Replica Management Framework

Dong Li

Supervisor&Examiner
Associate Professor Vladimir Vlassov
(ECS/KTH)

Master of Science Thesis
Stockholm, Sweden 2006
ICT/ECS-2006-50

# Abstract

Data (file) replication is an important technology in the Data Grid that allows reducing access time and improving fault tolerance and load balancing. Typical requirements to a replica management system include QoS (efficiency) specified, for example, as an upper bound on Round Trip Time; scalability, reliability, self-management and self-organization, ability to maintain consistency of mutable replicas.

This thesis presents design and a prototype implementation of a scalable, autonomous, service-oriented replica management framework in Globus Toolkit Version 4.0 using DKS, which is a structured peer-to-peer middleware. The framework offers scalable and self-organizing replica management service provided and consumed in a P2P network of Grid nodes. The framework uses the ant – a social insect paradigm – and techniques of multi-agent systems for collaborative replica selection. To validate and evaluate the approach, a system prototype has been implemented in the GT4 environment using the DKS P2P middleware. The prototype has profiled and tested on a computer cluster.

## Acknowledgements

We are grateful to Professor Vladimir Vlassov for his help in the design and thesis writing; to Mr. Konstantin Popov and Ali Ghodsi for help with the performance evaluation and setting up test bed.

# Table of Content

# Table of Figures

# Table of Tables

# 1 Introduction

The Grid, first brought forward by Ian Foster and Carl Kesselman, is the computing and data management infrastructure that provides us with the ability to link together resources as groups of complementary parts to support the execution of applications. The essences of the Grid lay in three folds [1]. Firstly it is distributed. Its working environment could be heterogeneous and dynamic. The second, it should use standard open general–purpose protocols and interfaces. The third, nontrivial qualities of service should be delivered to meet complex user demands.

The Grid can be divided into two categories according to application cases. One is the *Computational Grid*, where large compute facilities are shared and computing intensive jobs are sent to be executed at remote computational sites. The other is the *Data Grid*, which emphasize applications that consume and produce large volumes of data. As far as the Data Grid is concerned, a typical application case can have a large number of data files distributed and replicated all around the globe. How to manage all these replicas is not an easy task. We should consider heterogeneous and large scales such as wide area network. Here when we talk about the replica, we mean the replica in the granularity of file level. It is put in the shared storage space and exports to the outside user.

The principle functionality of replica management systems for the Data Grid is to maintain and furnish information related to file replicas. Additionally we prefer that it provides good scalability, some data consistency [2] among replicas according to the application situation, and optimized replica distribution for reducing access time and so on. A typical replica management contains the following basic elements.

- *Information services*, which could include replica location service, user access history, metadata catalog and so on. For some services, such as metadata catalog, centralized services could be accepted. But for others, such as Replica Location Service (RLS), they are expected to be distributed to provide scalability and fault-tolerance.
- *Data transfer service*, which provide file transfer among remote sites. Advanced data transfer service, such as Reliable File Transfer (RFT) [7] supports data transfer status information, transfer statistics, resuming transfer from the checkpoint, etc.
- *Security mechanism*, including authentication and authorization for remote users and secret, security communication. It may be required to provide securities for the third party transfer, a very common use patterns in the replica management.
- *Data consistency*. There are different requirements for it in the Data Grid according to the application case. For many scientific datasets accessed in a read-only manner, they do not need data consistency at all. For writable files, different levels of consistency may be considered.

Higher-level replica management components are constructed on top of these basic elements, such as Replica Management Service (RMS) in the EU DataGrid [52], and

Data Replica Service (DRS) [6]. This thesis work is such kind of higher-level replica management system. With the help from the GridFTP [9], two components providing information services (Replica Location Service and Node Location Component) and a data consistency mechanism, our system provides the optimized replica selection for users.

Regarding the security element, this paper will not pay much attention to it. In our system we take advantage of simpleCA [56] in the Globus toolkit 4 (GT4) [3] to set up Globus Grid Security Infrastructure (GSI) [5] for secret, tamper-proof, delegatable communication between services. In addition [4] provide a survey on decentralized security and the consequences of decentralized security among the Grid sites. Those methods described in [4] will work as good alternatives.

In the data grids, especially those lied in the wide area network, both network traffic and node status are dynamic. These presents challenges to the users who need to manage large amount of files. In addition the management work, including the replica life time management, the category maintaining, the data consistency among replicas, tend to be complex and overwhelming for the people. Therefore we expect our replica management system complies with the autonomic computing concepts. We expect components manage themselves according to polices set by the user in advance.

The autonomic computing first invented by IBM [11], refers to the computing systems that can manage themselves given high-level objectives from administrators. The fundamental building blocks includes: self-configuration, self-healing, self-protection, self-optimization and self-learning. We will show how our system works with these principles and how new Grid nodes integrate as effortlessly as a new cell establishes itself in the human body.

In our replica management system, we use a social insect paradigm called *ant*, which is a *complex adaptive system* (CAS) [57]. CAS is commonly used to explain the behavior of certain biological and social systems. It usually consists of a large number of relatively simple autonomous computing units, or agents. With the help from ant we can deal with the dynamics in the large scale Data Grid.

Most of the thesis work is implemented in the GT4, which supports Web Service (WS) Resource Framework (WSRF), WS-addressing, WS-Notification, and other basic WS specifications [8]. The WSRF proposal is a further evolution of Open Grid Services Infrastructure (OGSI). It presents the functionality missing from Web services from a Grid perspective [10]. With the WSRF implementation in the GT4, our services can expose and manage state associated with services, back-end resources, or application activities.

It is believed although peer-to-peer computing, grid computing, and web services arose independently, they are similar in spirit and purpose. The thesis work takes advantage of these technologies and we can see how they coalesce and cooperate in the later chapters.

## 1.1 Goals and Expected Results

The purpose of this project includes:

(1) Study of web services and Grid, and structured overlay network with DHT functionality;
(2) Development, implementation and evaluation of one replica management system. Its main feature should emphasize scalability and autonomous.

Expected results of this project include:

(1) A survey of combination between Peer-to-Peer technology and the Data Grid;
(2) A survey of related work in replica management in the Data Grid.
(3) Analysis and comparison of some (most relevant) approaches observed in (1) and (2);
(4) A vision of a scalable autonomous replica management system for the Data Grid;
(5) Architecture (including different services and protocols) of a replica management system, its description;
(6) A prototype of a replica management system;
(7) An evaluation procedure (parameters, test bed, applications), results of evaluation and their analysis.

## 1.2 Structure of the Thesis

Chapter 2 explains the technology background related to this work. A survey for the current P2P middleware, including structured and unstructured, is given. The Distributed K-ary System (DKS), which is a structured-overlay Peer-to-Peer (P2P) middleware used in our system, is described in more details. We also give an overview of the GT4 at the component-level. Then the concepts of autonomic computing are explained. At the end of the chapter, we try to illustrate the relationship between the OGSI, WSRF and Grid.

Chapter 3 is the core of this thesis paper. It describes the system designs in great details. The replica selection with the ant is presented as a use case.

Chapter 4 summaries related work done in the replica management system. It is explained in the following five aspects, i.e. RLS, data consistency, data transfer, security issues and higher-level replica management.

Chapter 5 talks about the implementation details.

Chapter 6 describes performance evaluation which emphasizes on the effects of different design strategies and file access patterns on the system response time and efficiency.

Chapter 7 supplies a conclusion for our work and envisions future work which includes possible system design improvements.

# 2 Background

## 2.1 Peer-to-Peer (P2P) Computing

P2P systems provide a way to harness resource for a large number of autonomous participants. In many cases, they are distributed Internet applications and form self-organizing networks that are layered over the top of conventional Internet protocol and have no centralized structure. The environment where P2P lies usually has millions of users, dynamical network traffic and variant user membership. Therefore P2P in general is characterized by massive scalability and global fault-tolerance.

P2P systems and Grid systems have spirits in common in that both systems have arisen from collaboration among users with a diverse set of resources to share. It is believed, as Grid systems scale up and P2P techniques begin to capture shared use of more specialized resources, and as users are able to specify location, performance, availability and consistency requirement more finely, we may see a convergence between these two techniques [12]. Figure 1 gives us a possible combination case. In fact we've seen many practical examples showing this kind of convergence [13] [14]. This thesis work also use Distributed K-ary System (DKS), a structured P2P system in a Grid system.

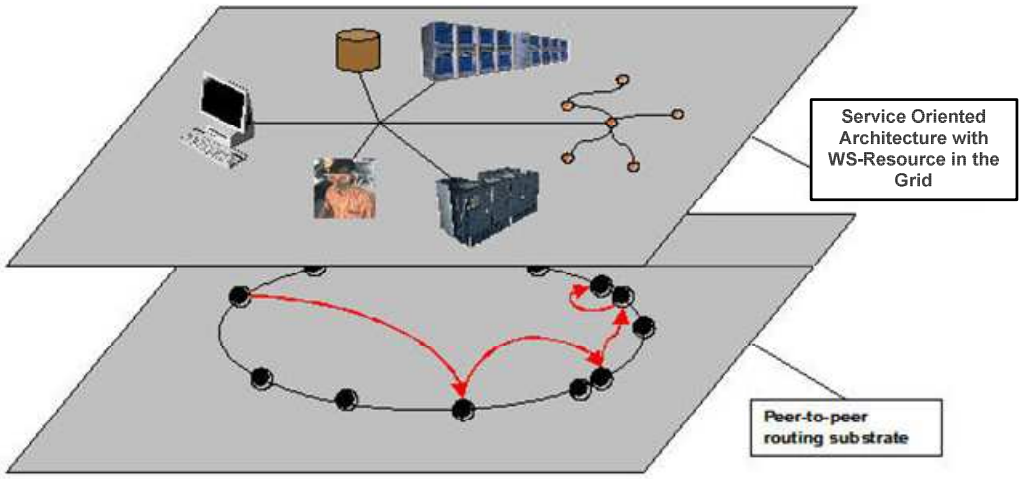


**Figure 1 A P2P Grid System based on Figure 6 from [12]**

P2P systems can be categorized into two sorts according to the routing substrates [12], structured P2P systems and unstructured P2P systems. The essential difference is that whether each peer maintains organized neighbors so that location of a piece of content or a node can be determined.

### 2.1.1 Unstructured P2P

In each P2P system, no matter structured or unstructured, a node maintains information associated with parts of participating nodes. The difference is that unstructured P2P systems nodes tend to replace their entries if it detects that the node in question has failed. It is more flexible in their neighbor selection and routing mechanisms. This means the topology of the network grows in an arbitrary, unstructured manner; it becomes difficult to bound the maximum path length and guarantee even connectivity between groups of nodes. In addition this system exhibits preferential connection tendencies toward highly connected nodes; therefore node failure of these highly connected nodes may be very sensitive.

There are three search mechanisms [15] in the current unstructured P2P networks, which are (1) flooding searches, (2) random walks and (3) identifier search. In a flooding search, when a node receives a query, it simply forwards the query to all of its neighbors. Gnutella [53] uses this kind of method. A query in Gnutella can overwhelm the network with messages. So it tends to be inefficient and waste bandwidth. Although we can set Time To Live (TTL) value to limit message life time, finding a appropriate TTL is not easy. Random walks [16] are simple. It means randomly walks the network querying each node it visits for the desire object. This method has long response time for resolving a query but it does reduce the number of messages and save bandwidth. Identifier search is based on the Bloom filters [17], which are essentially a potential function that guides the walk to allow the search to converge toward the object. It generates few messages than the above two messages and is faster than a random walk. A Bloom filter is a compact representation of a large set of objects that allows one to easily test whether a given object is a member of that set.

Although unstructured P2P systems have the shortcoming mentioned above, it does has big flexibility and support applications that require multi-attribute and wild card searching, which structured P2P are not suitable for these applications. In [15] the authors show that carefully constructed unstructured overlay can resolve this type of search within short hops for large networks and low replica replication ratios.

### 2.1.2 Structured P2P

Currently structured P2P systems seem to have the same meaning as Distributed Hash Table (DHT). They organize their peers in a way that any node can be reached in a bounded number of hops, typically logarithmic in the size of the network. In general, peers (nodes) identification and key items have the same address space. Each node is responsible for storing a range of keys and corresponding objects. By looking up a key, we can find the identity of the node storing the object paired with that key. A key is usually generated by hashing the object name. The DHT nodes are organized into overlay network where each node has several other nodes as neighbors. As a lookup request is issued from one node, the lookup message is routed along the overlay network to the node that is responsible for the key. There are many structured P2P systems. The Differences among them lie in the routing algorithm or the way peers organized.

Chord [18], Tapestry [19], Pastry [20], and SkipNet [21] are typical structured P2P system. In Chord both node identifiers and object keys lie in a one-dimensional circular identifier space with modulo $2^m$. Chord hashes one node's IP address and port to get a unique m-bit identifier for the node. A ring topology is set up based on all nodes identifier in the circular space. Each object owns an object key which is also a unique m-bit identifier. These object keys are allocated to nodes according to consistent hashing, which means key *k* is given to the first node whose identifier is equal to or follows the identifier of *k* in the circular space. Each node maintains two sets of neighbors, i.e. its successors and fingers. The finger nodes are distributed exponentially around the identifier space. To describe its routing mechanism, let's imagine a node *n* would like to look up the object with key *k*. This lookup request will be routed to the successor node of key *k*. If it is too far, node *n* will forward the request to the finger node whose identifier most immediately precedes the successor node of key k. This process could be repeated for many times until the successor node receives the lookup request for the object with key *k*, finds the object locally and return the result to node *n*. In addition Chord has some mechanism to achieve load balancing and fault tolerance, and maintain ring topology correctly by running stabilization protocol.

In Pastry, each node has a unique *nodeId* (identifier). When presented with a message and a key, a Pastry node can efficiently route the message to the node with a *nodeId* that is numerically closest to the key, among all current alive Pastry nodes. The "closest" means two numbers has matched prefix that is as long as possible. Each Pastry node maintains a routing table, a neighborhood set and a leaf set. The routing table in each Pastry node has $\log_2{}^b N$ rows, where *N* is the number of nodes and *b* is a configuration parameter with typical value 4. Each row has $2^b$-1 entries. The entries at row *x* and column *y* represents the node which shares with the current node an *x*-digit prefix, while the *x*+1 digit is *y*. Each entry in the routing table refers to any node whose *nodeId* has the appropriate prefix. Each entry also maps the *nodeId* to its IP address. Only the nodes which are likely to be close to the current node are selected to put in the entry. If no node is suitable, the entry will be left blank. The leaf set contains the entries of the nodes numerically closest to the current node, which includes $2^b/2$ nodes with larger *nodeId*s and $2^b/2$ nodes with smaller *nodeId*s. Pasty takes into account network locality. It seeks to minimize the distance messages travel, according to a scalar proximity metric like the number of IP routing hops. Whenever a node A is contacted by another node B, node A checks whether B is a better candidate for one of its entries according to the proximity metric. Pastry employs the locality information in its neighborhood set to achieve topology-aware routing, i.e. to route messages to the nearest node among the numerically closest nodes.

Tapestry is very similar to Pastry except the way how it manages replication and how it maps keys to nodes in the sparsely populated id space. For fault tolerance, Tapestry inserts replicas of data items using different keys. There is no leaf set and neighbor nodes are not aware of each other. If there is no entry for a node that matches a key's *n*th digit in a node's routing table, the message will be forwarded to the node with the next higher value in the *n*th digit, found in the routing table. This is called surrogate routing, i.e. mapping keys to a unique live node if the node routing tables are consistent.

Neither Chord Pastry nor Tapestry provides control over where data is stored and no guarantee that routing paths remain within an administrative domain whenever possible. SkipNet [21] is a scalable overlay network that provides controlled data placement and guaranteed routing locality by organizing data primarily by string names. SkipNet has two separate ID spaces, i.e. string name ID space and numeric ID space. The former consists of node names and item identifier strings. The latter consist of hashes of item identifier. Each node in a SkipNet system maintain $O(\log N)$ neighbors in its routing table, where $N$ is the number of nodes. A neighbor is said to be at level $h$ with respect to a node if the neighbor is $2^h$ nodes away from the node. This scheme has something in common with the fingers in Chord. At level $h$ there are $2^h$ rings where each has $n/2^h$ nodes. A search for a key begins at the top-most level of the node by seeking the key. It proceeds along the same level without overshooting the key, continuing at a lower level if required, until it reaches level 0. Content and routing path locality in SkipNet is enforced by suitably naming the nodes and incorporating node's name IDs in object name. In addition, SkipNet takes network proximity for both name ID and numeric ID routing into consideration.

The above structured P2P overlay network has topology-aware routing. Pastry and Tapestry contain information on which nodes are close to each other according to specific metrics, such as network latency. SkipNet has further control over data placement. In many cases, replica management system in the Data Grid tends to pick up appropriate replica positions which are the "closest" to a job execution site (the meaning of "closest" depends on different use cases). We believe if proximity information in the structured P2P, like Pastry, Tapestry and SkipNet, is fully explored for the Data Grid, they could give us more flexibility and convenience in the replica selection and management.

### 2.1.3 Distributed K-ary System (DKS)

DKS is a structured peer-to-peer middleware developed at KTH and Swedish Institute of Computer Science in the context of the European project PEPITO. The DKS is based on Chord and is a typical DHT. The routing table in each node maintains $\log_k(N)$ levels, where N is number of nodes in the network and $k$ is a configuration parameter. Each level contains $k$ intervals with pointers to the first node encountered in the interval. This kind of structure looks like a spanning tree, while Chord is just a binary tree. The lookup is resolved by following a path of the spanning tree. This ensures logarithmic lookup path length. DKS organizes peers in a circular identifier space and has routing tables of logarithmic size. Each node is responsible for some interval of the identifier space, just like Chord. DKS also self-organizes itself as nodes join, leave and fail.

Besides these typical DHT functionalities, DKS works more. It has several characters that separate it from others. The first is the different topology maintenance mechanisms [23], including correction-on-change and correction-on-use. Correction-on-change means whenever some changes happen in the topology, instead of depending on periodic stabilization to correct the topology, the correction should be happened immediately to reflect these changes. In other words, as a node join, it should immediately notify every node that point to it; as a node leaves, it should immediately notify every node pointing to it to point to its successor; as a node fails, the detecting node finds its successor which in

turn notify all nodes pointing to the failed node to point to it. This requires a node sense who is pointing to it. For a non-fully populated topology, a theorem [23] gives us some hints about which intervals we can find pointing nodes. Correction-on-use is kind of lazy compared with correction-on-change. When routed, messages piggyback information about the nodes neighborship. Then receiver can calculate if the pointer should be corrected.

Secondly, DKS use symmetric replication [22] to enable information backup and concurrent requests. Chord doesn't back up information stored in the system. So if a node fails, information it contains is lost. In order to improve fault tolerance, we would like to have several replicas for the same information. In this case, if we employ the nodes in the Chord successor lists to back up information, the master replica node becomes performance bottleneck and brings potential security problems. Therefore DKS abandoned successor lists. Instead it takes a mechanism called "symmetric replication". It partitions the identifier space into $m$ equivalence classes such that the cardinality of each class is $f$, where $f$ is the replication factor. Each node replicates the equivalence class of every identifier it is responsible for. So for every identifier $i$, there exists $f$ different identifiers in its equivalence class and every node knows this partitioning scheme. With this brand-new method, replicas can be accessed randomly, so DKS provides better load-balancing, stronger robustness than Chord and provide better security. Furthermore, if locality and proximity information is added for choosing replicas, shortened response time can be expected. With this method, it is also easier for us to delete some information stored in the DHT, because replica positions for each item are all deterministic and known by each node. This is important for choosing DKS to build the replica location service, which may need to modify (including deleting and updating) replica location information frequently.

Thirdly, in DKS join or leave operations are all locally atomic. This means join or leave operations never leave with failed or unfinished leftover in the system. By doing so, we guarantee lookup will always succeed. In DKS each node has three states including *gettingIn*, *inside*, and *gettingOut*. All operations on peers are defined for these three states. As a new node is inserted, the inserted point queues other insertion requests and doesn't exit. As a new node is leaving, the operation point should queue other requests and doesn't exit.

Finally DKS implements broadcast and multicast mechanisms [24] [25] [26]. Multicast in DKS proceeds in parallel and only affects specific multicast group members. Each multicast group can be tailored to meet specific requirements. All nodes are members of an instance of DKS($N$, $k$, $f$), where $N$ is number of nodes, $k$ and $f$ are the parameters mentioned in the above. Whenever a multicast is required, a node firstly creates a DKS instance and makes the group with the characteristics ($H_g$, $N_g$, $K_g$, $f_g$) according to the requirements and then achieve multicast by broadcasting within the group. As far as broadcast algorithm is concerned, it goes from one interval to another. Intervals are covered in counter-clockwise direction. The word "limit" is introduced to refer to an operation delegating intervals to responsible nodes. We commit multicast within an interval and change the "limit" after a multicast message is sent.

From the above, we can see DKS is a typical structured P2P system. Besides its basic function like a common DHT, it has symmetric replica mechanism for information backup, so the replica position is deterministic which is good for deletion or updating operation. Its topology maintenance is efficient and saves bandwidth. In general DKS provides a very good middleware for our upper replica management. We believe if given more time, we can improve our replica management system for the Data Grid by fully exploring it.

## 2.2 Globus Toolkit 4

The Globus Toolkit (GT) is an open source software toolkit used for building grids [3]. It is being developed by the Globus Alliance and many others all over the world. It supports the development of service-oriented distributed computing applications and infrastructure. In essence, GT is a set of libraries and programs. Within a common framework, core GT components address many issues including resource discovery, data movement, resource access, resource management, security and so on. These issues are indispensable for developing advanced functions or conducting science researches such as biology molecule formation and high-energy physics data analysis. Therefore GT provide a good platform for others application and tools to be built on or interoperate with. From GT3 which supports Open Grid Service Infrastructure (OGSI) to GT4 which supports Web Service Resource Framework (WSRF), GT makes extensive use of web services to define interfaces and structure of its components. In section 2.4, we will discuss more about the relationship between stateful web services and Grid.

GT4 consists of three sets of components. The first is the set of service implementation. Most of them are Java Web services except that GridFTP, MyProxy, RLS and Pre-WS GRAM are implemented in C language. The set includes GRAM for execution management, GridFTP and RFT for data movement, OGSA-DAI for data access, RLS and DRS for replica management, Index, Trigger and WebMDS for monitoring and discovery, MyProxy, Delegation and SimpleCA for credential management. The second set is three containers, i.e. Java, Python and C containers for hosting user-developed services. These containers provide implementations of security, management, discovery, state management, and other mechanisms frequently required when building services. They extend open source service hosting environment with support for a range of useful Web service specifications [27], including WSRF, WS-Notification and WS-Security. The third is a set of client libraries. These allow client programs in Java, C and Python to invoke operations on both GT4 and user-developed services.

In the rest of this section, we will say a little more about GT4 structure at the level of function. They are more or less related to our work. With the following description, we expect to give this paper a bit self-contained. In addition Figure 2 gives a GT4 architecture schematic which is adapted from [27].

**Data Movement and Access**

GT4 includes the implementation of GridFTP specification, which includes libraries and tools for reliable, secure, high-performance memory-to-memory and disk-to-disk data

movement. Currently more and more data movement services or applications are based on it, such as Reliable File Transfer and Data Replication Service (DRS) [28].

Replica Location Service (RLS) is a scalable system for maintaining and providing access to information about the location of replicated files and datasets. The GT4 uses a framework called Giggle [29]. We will talk about RLS a little more in the section 4.1. RFT provides the information and management of multiple GridFTP transfers. DRS is sort of upper service, since it combines RLS and GridFTP to provide for the management of data replication. The Globus Data Access and Integration (OGSA-DAI) tools provide access to relational and XML data.

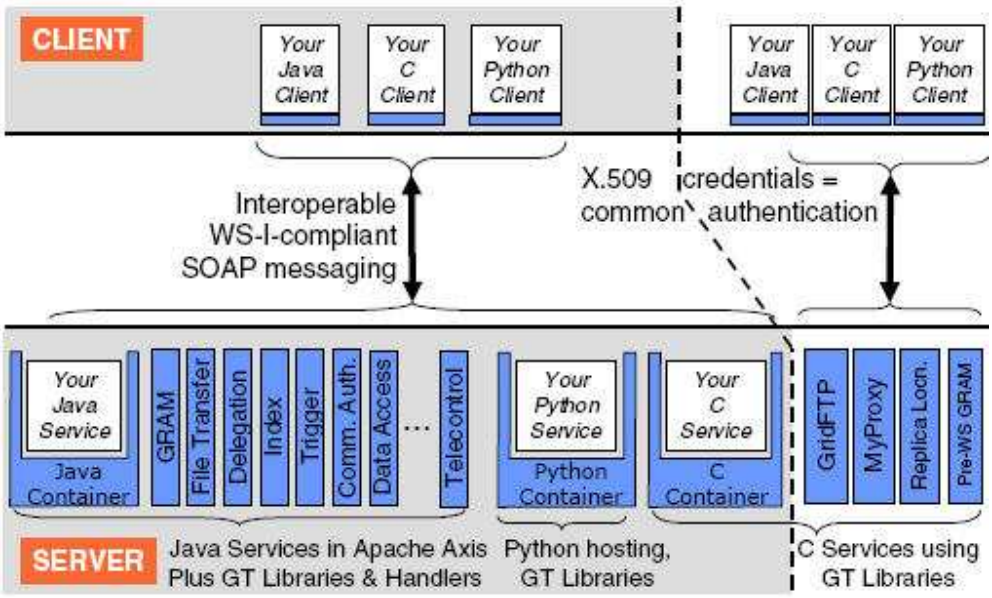


**Figure 2 GT4 architecture schematic, including many components**

**Monitor and Discover Services and Resources**

In GT4, there are two mechanisms to detect problems or identify resources or services with desired properties from distributed information sources. (1) Associating XML-based resource properties with network entities and accessing those properties via query or subscription operation according to WSRF and WS-Notification specification. Without much more labors to do in users' services, users can incorporate this into their own developed service. Furthermore all registered services or containers can be organized into a hierarchical structure, which is easily managed. (2) Aggregator services, which collect state information via aggregator source. These sources could not only be those supporting WSRF/WS-notification interfaces, but also be external software components, such as Hawkeye [30] and Ganglia [31]. They use common configuration mechanisms to maintain information about which aggregator source to use and its associated parameters. Each registration in these services has a lifetime, i.e. they are self-cleaning. There are two aggregator services provided by GT4, named Index service and Trigger services. The index service collects information and publishes that information as resource properties. Clients use the standard WSRF resource property query and subscription/notification

interfaces to retrieve information from an Index [32]. The Trigger Service collects information and compares that data against a set of conditions defined in a configuration file. When a condition is met, or triggered, an action takes place [32]. In this thesis work, we use the default index service in the GT4 Java container.

**Security issues**
Although our replica management system doesn't take security issues into consideration, it is an important thing for the Grid, especially when resources and users span multiple locations. GT4's highly standards-based security components implement credential formats and protocols that address message protection, authentication, delegation and authorization [34].

GT4 support both message-level security and transport-level security. Message-level security supports the WS-Security standard and the WS-SecureConversation specification to provide message protection for SOAP messages. However message-level security implementations have poor performance. This lies in two points. One is that the XML Signature design which is used by SOAP applications introduces a number of complex processing steps [33], such as canonicalization and XPath filter, leading to performance and scalability problems. The other is implementation issues [34]. Transport-level security is based on X.509 credentials. This is the default security mechanism and faster than message-level security. It is believed Transport-level security is just a temporary solution. Message-level will replace it sooner or later, because it complies with the WS-Interoperability Basic Security Profile.

In the default configuration, all services and uses share a common certificate authority and have a X.509 public key credential. Then two entities validate each other's credentials and set up a security channel for purpose of message protection. They may create an attenuate proxy certificate to allow another component to act on a user's behalf to authenticate to the target within a limited period of time. This security work flow has more or less centralization factors in the design. As an alternative, distributed security can be used [4].

**Execution Management**
Sometimes users need to dispatch individual tasks to computational cluster or, deploy service and control its resource consumption, or use the Message Passing Interface to schedule subtasks across multiple computes, etc. The GT4 addresses these use situations by giving us Grid Resource Allocation and Management (GRAM). It is a web service interface for initiating, monitoring and managing the execution of arbitrary computations on remote computers. In our thesis work, we didn't use GRAM. But we think GRAM could be added into our system later to watch and control local resource consumption.

## 2.3 Autonomic Computing

Autonomic computing, invented by IBM, is an idea borrowed from biology field. A component which has the characters of autonomic computing in a system is just like a normal heart in human body. Normal heart beats regularly and it doesn't need any intervene of brain. You or the nature may set policies or rules for the heart in advance.

Afterwards, it just works according to these policies or rules. It cooperates with other organs and has self-* properties, such as self-configuration, self-optimization, self-healing to some extend and so on. In other words, it is totally autonomic, which is very reasonable. The whole human body is complex and consists of many parts. We can't imagine if all of them depend on brain's guides to do everything --- either brain will be exhaustive, or it will become the "bottleneck" and react pretty slowly.

In the Data Grid, according to requirements and situations of science research, large amount of files need to be managed. They lie around in a large scale, even in the global scale. The statuses of the whole system are changed, such as a replica is created or removed, a Grid site is crashed. It is hard to monitor and manage them manually. This situation is just like the brain we mention above. Therefore when we design the replica management system, we try to make it conforming to the "spirits" of autonomous computing, i.e. after user deploys parameters and requirements for replica management, he doesn't care about the later issues any more. The system should work autonomously and need people's cares as less as possible.

There are four aspects of self-* properties as they are now and would be with autonomic computing [35].

- Self-configuration

It means the components follow high-level polices and automatically configure itself. It could adjust automatically and seamlessly.

- Self-optimization

Components and systems continually seek opportunities to improve their own performance and efficiency.

- Self-healing

When software or hardware problems occur, system automatically detects diagnoses and repairs them.

- Self-protection

System has the security mechanism that automatically defends against malicious attacks and failures. It uses early warning to anticipate and prevent system-wide failures.

We will see in the chapter 3 how our system automatically works according to these properties.

## 2.4 OGSI, WSRF and Grid

Many current Grid infrastructures are moving towards service-oriented architecture (SOA). Web services, working as a standard for a particular set of XML-based technologies, are heavily replied on to build SOA and used widely in the GT. They provide great flexibility to set up loosely coupled components (services) and dynamically compose them. Their natures are distributed and can work pretty well in the Grid situation which is dynamic and heterogeneous.

Web services are typically implemented by stateless components. They are usually modeled as stateless message processors that accept request messages, process them in some fashion and formulate a response to return to the request. However sometimes we

need pieces of information in order to properly process the request. These are called *states*, which means "a set of persistent data or information items that have a lifetime longer than a single request/response message exchange between a requestor and the web service" [36]. The states which are bundled together are named as *stateful resource*.

The previous applications happened to deal with stateful resources in different manners, although they showed the same relationship between web services and state. This situation increased the integration cost between these applications and limited the reusable of middleware. Therefore Open Grid Service Infrastructure (OGSI) and WSRF have been proposed to formalize and standardize the relationship between web services and state.

The OGSI, developed by the Globus Alliance, was the first attempt to associate web services and their states in a standard way. It was expected to adopt and exploit web services and made Grid services more widely accepted. OGSI introduces the notion of Grid Service which is a variant on the web service concept. Grid service defines a standard set of operations that can be performed on any Grid services. It is regarded as "an attempt at a component model for web services" [37]. It is just like an Object class in object-oriented languages [37].

However OGSI was not accepted. It made too aggressive use of WSDL and XML. Its extension to WSDL, GWSDL didn't allow general WSDL tools to build OGSI system. It also confused many people by blurring the notion between stateless web services and state resources [37]. Finally it defined too many concepts in a single specification. Many of them should be composed from others instead of defined individually. These significant shortcoming prevented the widely support for Grid infrastructure. So the Globus Alliance took action again. They propose a further evolution from OGSI to the WSRF.

The WSRF can be regarded as a set of standards that are intended to unite the way Grid computing, system management, and business computing use web services. It answers questions that are of interest to applications, including [36] (1) how a stateful resource is identified and referenced by other components n the system; (2) how messages can be sent to the stateful resource in order to act upon or query its state value; (3) how the stateful resource is created, either by an explicit Factory pattern operation or an operation within an application; (4) how the stateful resource is terminated; (5) how the elements of the resource's state can be modified. WSRF functionality is separated into five independent specifications that define the normative description of the web service resource in terms of specific web services message exchange and related XML definitions [58]. These specifications are summarized in the Table 1 which is from [58].

The changes from OGSI to WSRF are primarily syntactic but also represent some useful progress [37]. In OGSI, stateful resources are called Grid services; while in WSRF, they are called WS-resources. Although they have the different names, they have the same ability to create, address, discover, and manage stateful resources. In addition WS-Addressing is used and XML schema is used less. Web services community new progress,

such as WSDL 2.0 is also taken in WSRF. Given WSRF, in particular WS-Notification, it is easier to define information service components for discovery, monitoring, fault detection and so on. Currently these changes get more support in the web service community for Grid infrastructure. In fact our system is taking advantage of this change in the WSRF. Some services in our system have stateful resources. With GT4 which supports WSRF, we can easily develop satisfying information service.

**Table 1: Five normative specifications defined in the WSRF**

| Name | Description |
|---|---|
| WS-ResourceLifeTime | Mechanisms for WS-Resource destruction, including message exchanges that allow a requestor to destroy a WS-Resource, either immediately or by using a time-based scheduled resource termination mechanism. |
| WS-ResourceProperties | Definition of a WS-Resource, and mechanisms for retrieving, changing, and deleting WS-Resource properties. |
| WS-RenewableReferences | A conventional decoration of a WS-Addressing endpoint reference with policy informatioin needed to retrieve an updated version of an endpoint reference when it becomes invalid. |
| WS-ServiceGroup | An interface to heterogeneous by-reference collections of web services. |
| WS-BaseFaults | A base fault XML type for use when returning faults in a web services message exchange. |

# 3 Design

This section gives an overview of the design of our replica management system. We discuss the main components of our system and identify the functionalities and interdependencies of the components. Here the "component" refers to a function unit, such as location information or replica selection. It may include one or more than one services. The "service" refers to the web services with or without resource properties.

## 3.1 System Design

Our replica management system is intended to manage and place replicas according to user specific QoS requirements and access records. Here the "user" refers to a job execution node. In the Data Grid (see Figure 3), after a job is scheduled by a resource broker to a node where workload is appropriate, this node is a job execution node. It may need a large amount of data files for computing. For each needed file, our replica management system is able to provide a suitable replica location to minimize file access time according to the user Round Trip Tim (RTT) requirement. Figure 4 illustrates the components and services of our replica management system in a Data Grid node.

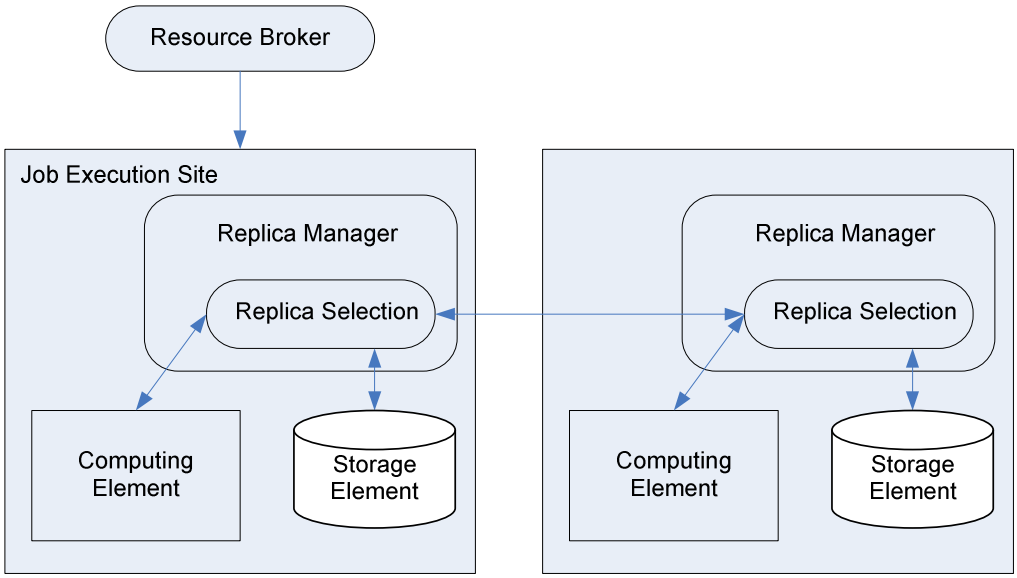


**Figure 3 DataGrid Architecture**

In general the system is designed according to the following Principles.

- Oriented towards large scales and having good scalability
- Compliance with the spirits of autonomous computing
- Adoption of the web services and WSRF standards to promote interoperability
- Achieving good fault tolerance.

The system has three reusable lower-level components, i.e. *Location Information*, *Data Consistency* and *Data Transfer* and two higher-level components called *Replica Selection*

*(RS)* and *Statistics* which act upon the lower ones. They are explained in the following sections.

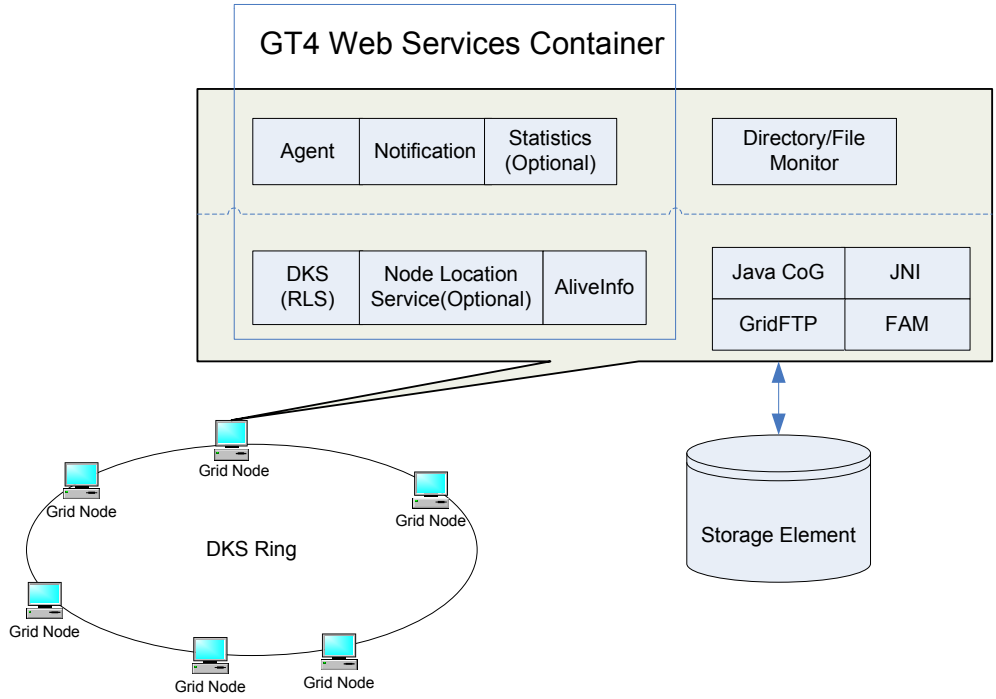


**Figure 4 Main Components of Replica Management System**

## 3.1.1 Location Information Component

The Location Information provides information on replica location and all Grid node addresses in a virtual organization. By Grid node, we mean a Grid site linking to others and acting a role in a virtual organization. It consists of two parts, the Replica Location Service (RLS) and Node Location Component (NLC). The replica location service is built on top of the DKS, a structured P2P middleware which enables mutable data storage (see section 2.1.3). For each item (key-value pair) in the DKS, the key is the hash value of a unique file name. The value is all replica positions for this file. This service exports many APIs in the DKS as web service operations. Each Grid node has a local RLS. All RLSs are organized into a DKS ring and each has parts of the replica location information. The NLC includes Node Location Service (NLS) and "aliveinfo" service. It is based on the GT4 WS MDS Aggregator Framework with the Query Aggregator Source, where the NLS acts as aggregator sink and the "aliveinfo" is a WSRF service that registers its resource property to a service group in the NLS. Current resource properties in the "aliveinfo" include file lists in the storage element (see Figure 4) and node address. But it could include more in the future, such as workload of the local node and available storage space size etc. The NLS is based on the Index Service of the GT4 and collects position information (node address) of all registered living Grid nodes in a virtual organization. Each registered node position has limited life time and need to be refreshed by the corresponding node, otherwise it is removed. With the help from the aggregator framework, other service or application can retrieve positions information of living nodes

by querying the NLS. Whenever a Grid node starts up and would like to join a virtual organization, its "aliveinfo" service registers to the NLS and its local RLS seeks any registered node from the NLS as an entry point to join the DKS ring or starts a new DKS ring if there is no item in the NLS.

### 3.1.2 Data Consistency Component

The Data Consistency component takes care of keeping the replica set of a file consistent. It consists of the lower FAM and the upper application. At each node the File Alteration Monitor (FAM) [47] is used to monitor the statuses of a specific directory and all files within it. The FAM is an open source project and it detects changes to the monitored file system and relays these changes to the upper Data Consistency applications. Although currently we use the FAM only on the Linux, we expect this doesn't affect portability of this solution. The newest FAM (2.7.0) features a feature-based configuration script rather than an operating-system based script, which makes it easier to be built on other platforms. In addition the FAM is developed in C language. In order to access its library routine from the Data Consistency component, Java Native Interface (JNI) [47] is used. Taking advantage of the FAM, we observe when a file is changed and record it into a log. The Data Consistency periodically checks the log to see whether any changes are made to the monitored files. If so, these changes are sent to other replicas to make them consistency. When a replica is deleted or added, the upper application is notified by the FAM and updates corresponding replica location information in the RLS. This solution is a kind of epidemic approach (see section 4.2). It is flexible and independent of any application accesses type. However it can only guarantee weak consistency among replicas and may not work well for time-critical data. But in many cases, science data files are used for analysis and not changed frequently. This kind of consistency is enough. Furthermore we can configure time period of maintaining consistency to adapt to application requirements.

### 3.1.3 Data Transfer Component

The Data Transfer component exists on each node. It consists of a client for the GridFTP and one GridFTP server. The GridFTP supports third party transfer, which allows a user or application at one site to initiate, monitor and control a data transfer operation between the source and destination site for the data transfer. This is very important for the replica management.

### 3.1.4 Replica Selection Component

RS component includes two services, agent and notification. Each node has an obligatory agent service and an optional notification service. By "optional", we mean it is possible for the node to have no notification service. The agent service's responsibility lies in tow folds. Firstly, it is an entry point to replica selection, i.e. it can work as the "delegate agent" to receive the task from users and conduct selection work. Secondly it indirectly interacts with each other according to the "ant" algorithm and senses network environment to help delegate agent find appropriate replica. We will talk about these details in section 3.2. The notification service is used to inform users of the selected replica position for a requested file. Before a user delegates a request to an agent, it

creates a resource in a notification service, whose resource properties are the values associated with replica selection, such as file name, RTT requirement and selected replica position. The users can subscribe to resource properties or query these resource properties. As the delegate agent gets the final selection results, it changes the corresponding properties in this notification service and enables the user to receive asynchronous notification. For each replica selection, delegation agent and notification service could be in different nodes.

### 3.1.5 Statistics Component

The Statistics component is a service located in one node and therefore is centralized. It receives file change records from each Data Consistency component and replica selection results from the agent service. At each Grid node the directory monitor in Data Consistency component records changes times for each local file within a time (The changes for data consistency maintaining are not included) and periodically sends these records to the Statistics Service. Each delegate agent, after it gets selection results, will also report it to the Statistics Service. Here we assume that before accessing any file, the user uses the RS to decide file position. Therefore according to these access records, the Statistics Service could decide which replica may not be used beyond a threshold time and removes it. It also finds which file is popular by summing up access times for all replicas of this file. The files whose access times are beyond a threshold are called popular files. They may be proactively put to a place close to some Grid nodes with the help from RS component. These nodes need to submit their requirements to the Statistics Service in advance. We assume that previous popular files are also popular later. So it is highly possible that they are needed by these nodes. By putting potentially needed files close to nodes, we hope to reduce time for replica selection and replica movement which may be long for large size files.

## 3.2 Replica Selection

This section describes our replica selection method which is based on the ant algorithm. We will firstly explain how the ant works and then see how the ant is applied in the replica selection.

### 3.2.1 The Autonomous Ant

The ant algorithm simulates actions of the ant colony. It consists of a large number of relatively simple autonomous computing units, or ants. Their actions exhibit the characters of *emergent behavior*, i.e. although the interactions among them are simple, they can generate more complex and richer patterns than those produced by single ant in isolation. Resnick [49] describes an artificial ant following three simple rules: (i) wander around randomly, until it encounters an object; (ii) if it was carrying an object, it drops the object and continues to wander randomly; and (iii) if it was not carrying an object, it picks the object up and continues to wander. Although the rules seem simple, the ants conforming to them are able to collect small stuffs and group them into larger ones. In addition, in emergent systems, there is a method of communication called *stigmergy*. It means the individual parts of the system communicate with one another by modifying

their local environment. For example, ants communicate with one another by laying down pheromones along their trails. This can indirectly guide other ant's future behaviors.

The ant algorithm is self-organized, adaptive and distributed. It adapts to the large scale and dynamical environment. With the help from the P2P overlay, it can full explore participating nodes without the bothering of membership changes. Although the algorithm seems simple, it finishes users' tasks without any rules specific to variations in the environment, initial conditions and topology-aware. Currently the ant algorithm has been used in the Data Grid for load balancing [50] [51].

In our system when an agent service is "motivated" (there is operation in the agent service called motivation), it sends out ants. An ant walks from an ant container to another within configured number of steps, after which it returns to the delegate agent. An ant exists as a Java class containing the ant state or in other words all necessary data for the ant algorithm. The ant states include file name, current hop number, minimum RTT and corresponding position, delegate agent position, RTT requirement and user position. The ant container is an operation in the agent service implementing the ant algorithm. Each ant container, invoked by an ant, will determine the RTT between itself and the given user node. It updates the corresponding ant states if its RTT is shorter than the minimum. The ant carries its states and walks along the DKS ring. The behavior of an ant is determined by its current state and its interaction with the ant container. As you can see, our ant is somewhat different from the nature one. Instead of grouping objects, they collect node information and explore unknown space.

### 3.2.2 Replica Selection with the Help from the Ant

Figure 5 describes our replica selection method. The user begins by creating a resource in a notification service (1). Its properties include file name, QoS requirement and replica selection results. Currently we only consider one QoS parameter, Round Trip Time (RTT). This parameter reflects dynamic network environment and is important for predicting job execution time. In the future we can consider more parameters. The replica selection results are satisfying replica position and the RTT between the selected position and the user node. The user also subscribes to the resources properties of replica selection results. So whenever they are changed by the delegate agent, a notification is sent to the user node.

After (1), the user submits the parameters for replica selection to any agent service (2), which will be the delegate agent. These parameters are file name, QoS parameter (here is the RTT requirement) and the storage space size the user node can provide. To reduce file access time caused by network latency, the best place to put the file is user local storage element. However the user may have limited storage space. Therefore we ask the user to give the storage space size it can provide and it is up to the delegate agent to make a decision whether it is appropriate to put the file in the user node. Typically the user may choose the agent service located in the local place as the delegate agent. However it can also be in remote place and its position can be got from the NLS. In Figure5, the delegate agent is in the Grid node D.

After the delegate agent receives the parameters, it queries the RLS to find existing replica positions, which is the Grid node A in the Figure 5. From one of these positions, it gets the file size and decides whether the file could be placed in the user node. This step is not shown in the Figure 5. If the file is too large, the delegate agent contacts the agents in the existing replica positions to compute RTT to see if any one satisfies the RTT requirement (3) (4). If the satisfying location exists, its position and RTT are returned to the delegate agent that in turn changes resource properties related to the selection results in the notification service. If no one can satisfy the requirement, the delegate agent motivates the agents in the existing replicas (5).

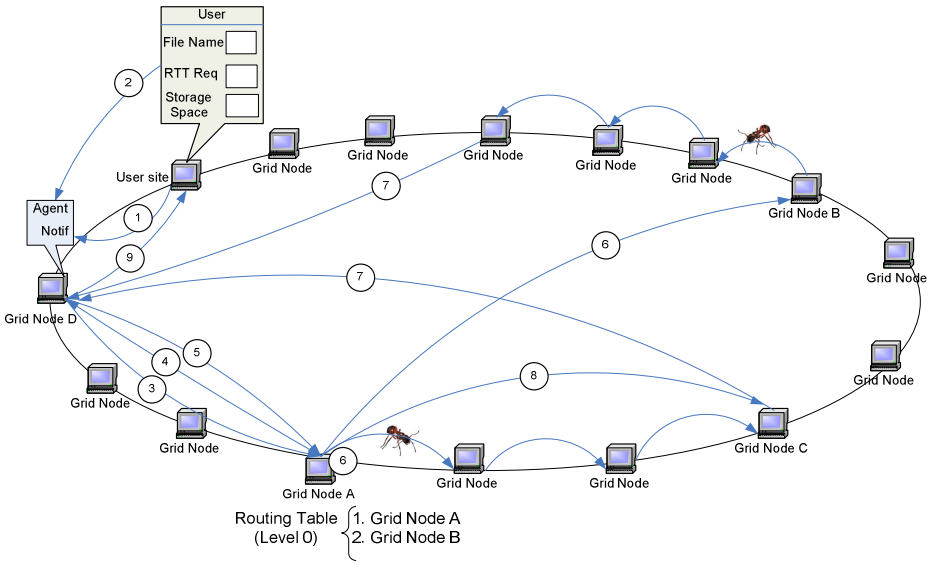


**Figure 5 Replica Selection**

Any motivated agent sends out the ants (6). Their destinations are the nodes in the first level (level 0) of DKS routing table for the nodes where the motivated agents live. In the Figure 5 we assume they are the node A and B. We choose the nodes in the first level of routing table to make ants being separated from each other as far as possible in the DKS ID space, so all Grid nodes could be fully explored. The ants walk along the DKS ring to collect information of each place they pass by and record the best position in their statuses. At each step, the default next destination for the ant is the successor of current node, just like what the Figure 5 shows. However the ant container can also check the information left by the previous ant. It may be lucky to find some previous records for positions where there was a shorter RTT between it and the user node. Then it chooses that place as its next destination. This is s a kind of *stigmergy* behavior, i.e. the "smell" or information left by previous ants guide later ant action. In the performance evaluation, we will see how this model shortens ant exploration time. After walking fixed steps (in the Figure 5 there are three steps), all ants go to the delegate agent where the best position will be chosen (7). As an alternative, we could ask the ant to go to the delegate agent as soon as it finds any satisfying position. However this position may not be the best position the ant can find and may cost users more access time. Therefore we would like to find the position that reduces access time as much as possible. Obviously this is done at the cost of asking ants to walk more steps. If the delegate agent discovers an

appropriate position, it creates a new replica in this position which is performed with the third party transfer in (8), registers it in the RLS, reports this selection result to the Statistics service and changes the resource properties associated with selection results in the notification service to inform the user (9). The replica is copied to this new position from the existing replica place where there is the shortest RTT between them. If the satisfying positions aren't found, the agent changes the resource properties without copy behavior.

## 3.3 Fault Tolerance, Scalability and Self-* Properties

As far as the fault tolerance of location information is concerned, we consider the case of node crashing. Whenever a node crashes, the replica location information stored in this node is missed and location information related to this node in the NLS and RLS need to be updated. For the replica location information stored in this node, the DKS has symmetric replica to enable information backup and concurrent requests. Even as the replica location information stored in this node is lost, other replicas still allow users to get information. In the NLS, the entry related to this node is removed when its life time expires and not refreshed by the crashed node. To update the replica location information in the RLS, we require that each query operation for the replica positions of a file, before it returns results, should get the newest node lists from the NLS and compare it with the query result. If any position doesn't exist according to the list, it is removed from the RLS. In this way the RLS is updated step by step. Of course this will add the overhead of getting the node lists to a query operation. To improve its performance, the RLS at each node could cache the node lists and refresh it periodically. Considering the usual case that the node crashing is not frequent, this caching method is acceptable.

Our replica management system has good scalability. Whenever a new node is added and joins the DKS ring, its storage space can be explored by ants. No extra efforts are needed to inform other nodes or ants except to register it in the NLS. The centralized NLS may affect the scalability a little bit. However since it is based on the Index Service of GT4, we can easily deploy more Index Service and organize them into a hierarchy way. This is not a problem. In addition, we use P2P overlay network for our RLS. This provides great flexibility and scalability for information storage. We believe that taking advantage of the DKS, autonomous ant and hierarchy location service, our system is able to work for the virtual organization containing a large amount of nodes.

Our replica management system is autonomous. It achieves self-configuration, self-healing, self-optimizing.

- Self-configuration

Following high-level policies, the replica management either picks up appropriate replicas or adjusts replica distribution. A new Grid node can be added into virtual organization easily. The new added storage space is explored and used automatically without any manual configurations. Weak data consistency is achieved and auto-maintained without any knowledge of application type and access patterns.

- Self-optimization

The replicas are optimistically placed to meet user specific QoS requirements. The useless replicas are removed silently. To reduce access time for files, The Statistics Service predicts future replica distribution according to the user access records.

- Self-healing

Whenever the node crashing happens, the system automatically detects it and updates information related to it. The other nodes are not affected by node failures. No manual healing for the system recovery is needed.

# 4 Related Work in Replica Management of the Data Grid

In this chapter we will give a survey for the work done in replica management. In general replica management is not a single software component. It may consist of several basic components (services). In the Chapter 1, we categorize them into 4 classes, i.e. information services, data transfer, security mechanism and data consistency. This chapter is presented according to these categories. Furthermore we summarize several high-level replica management systems and compare them with our system.

## 4.1 Replicas Location Service

In the Data Grid, replicas are used to provide shorter access time, fault tolerance and load balancing. One of the concerns for managing replicas is how to discover and register them. We call this service as *replica location service*, which is a general concept in this thesis paper and different from the specific one in the GT3 and GT4.

The replica location service needs to satisfy several requirements. The first and the most important is the scalability, which allows millions of location information to be stored in them. The second is dynamical membership maintenance. Here membership refers to a set of sites where replica location can be registered and discovered. The replica location services should be updated dynamically to reflect membership changes. The third is the fault tolerance. Location information cannot be lost if some sites storing replica location information are broken. This is indispensable; otherwise some files may happen to be disappeared from users' views although they still exist physically.

The replica location service in GT3 and GT4 is based on a parameterized framework called Giggle [29]. They define two terms called physical file name (PFN) and logical file name (LFN). PFN is the exact replica physical location. LFN is a unique identifier for the contents of a file which may have many replicas. Local Replica Catalogs (LRC) maintains the mapping between PFN and LFN at a place. Replica Location Indices (RLIs) collect the mapping information from LRC and are organized into a hierarchical distributed index (see Figure 6). Whenever user submits a LFN to a RLS server, she can get physical locations of all replicas for this LFN. To keep information consistency among LRCs and RLIs, information in RLIs need to be periodically refreshed by LRCs, otherwise it times out. LRCs send their new states by soft state update protocols. The states can be processed with Bloom filter compression scheme to save network bandwidth and storage space in RLI sites.

This framework is scalable in some sense. By organizing RLI in a hierarchy and distributed way, large amounts of mapping information can be stored. However it has a membership problem. LRCs and RLIs must be known in advanced for users. Their deployments are statically. RLI cannot be inserted into existing structure autonomously. It is also not easy to recover from any RLI failure.

P2P overlay networks are highly scalable and has no centralized membership management. They can tolerate membership dynamical changes, such as node leaving and joining. They are good replacement for the Giggle. So [38] [14] introduce different P2P overlay networks to work as the replica location service.

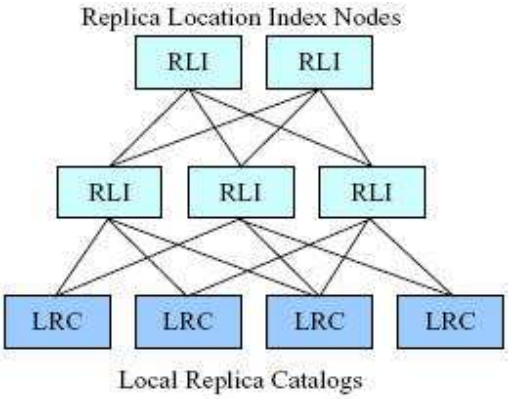


**Figure 6 Hierarchical RLS in the GT**

In [38] the author organizes all RLIs into a structured P2P overlay network, Chord. Taking advantage of this P2P, it is easier to handle cases such as RLI node leaves or is inserted into the existing ones. Scalability and fault tolerance are improved. However Chord doesn't provide the way to back up the data stored in each node. Users are at risk of losing data when a node crashes. Therefore the author uses a scheme called successor replication. It replicates the mapping stored on the *root* to its $k$ successors, where $k$ is the replication factor and is typically $O(\log N)$ for $N$ nodes. The root node is the node that is in charge of storing a specific mapping according to the Chord algorithm. In addition each mapping has a life time. It needs to be extended by the predecessor that owns the same mapping. In this way, the useless mappings can be cleared by itself. In order to maintain consistency among all replicated mappings, whenever replica location information is changed, LRC informs the root node immediately that in turn updates all the replicas. Furthermore the authors also made modification to the Chord to balance load. To evenly distribute the mappings among nodes, the factor $k$ is increased to store more mappings per node. To balance query load, *predecessor replication* is taken, i.e. replicating mappings in the predecessor nodes of the root node. This is based on the fact that before the query is routed to the root node, there is high probability that one of the predecessors of the root node is traversed. If the mapping is found locally in the predecessor, the result is returned without bothering the root node.

In [14] the authors design a replica location service oriented towards the future needs of a global Grid. This kind of Grid may have large number of members, frequently membership changes and good fault tolerances. The author introduces Kademlia [54] overlay as the replica location service. To enable mutable data storage in this structured P2P overlay, a simple data versioning scheme is introduced. It uses timestamp indicator for every key-value pair. *lookup* or *store* command update information with the latest values according to the data versioning scheme. For the case that frequent joins and departures in the P2P overlay causes continuously exchanging key-value pairs, this design is not affected. This is because the modifications to support mutable data storage

are based on operations, not node behaviors. The integrity of updated data is not affected by changed membership.

Our system also use a P2P overlay network, the DKS, as the RLS. For the details about the DKS, you can see section 2.1.3. Unlike the Chord, the DKS uses symmetric replication for information backup. We don't need to implement any "predecessor replication" to consider information loss. Besides, the position for the information backup is deterministic. This is easier for us to implement delete and update operation, or mutable information storage. For the information integrity, our scheme is just like the one in [14]. It is based on operations, i.e. query operation (see section 3.3), not on node behaviors. This can reduce too many key-value pairs exchanging due to possible frequent node joining or leaving.

## 4.2 Data Consistency

Data consistency in the Data Grids is not always required. Many scientific datasets are accessed in a read-only manner. Therefore some replica management systems, such as the Data Replication Service (DRS) in the [28] don't provide data consistency at all. In addition, considering possible large amount of data and locking overhead in the global scale, strict consistency is not practical, especially when you require a reasonable response time for accessing replicas over the WAN. Sometimes weak consistency can be acceptable if user requirements are satisfied.

Weak consistency is able to be established by having asynchronous replication. There are several commonly used solutions for asynchronous replication. They are summarized in [39]. Here we just borrowed their summaries. The first solution is primary-copy (or master-slave) approach. You can only modify the primary replica. Any write request towards other replicas is forwarded to the primary one. The primary is also responsible for updating and propagating the changes to the others replicas. This solution is taken in the object data stores Versant [40], ObjectStore [41] and Oracle products. The second is the epidemic approach. This method executes update operations locally first and then the sites communicate to exchange up-to-date information. This solution has very low degree of consistency and can only be applied for non time-critical data. The third solution is applied for the subscription and relatively independent sites. It allows a site to do local changes without the agreement of other sites. A site that explicitly subscribes to a data producing site is notified of the updates. A site that has not subscribed is itself responsible to get the latest information from other sites.

Models for replica synchronization and consistency in the Data Grid are given in [2]. The authors discuss data consistency levels delivered to Grid users based on database transaction theory including locking for establishing consistent data.

In our system, data consistency is not ignored. File storage in our design is non-database stores and doesn't provide transactional consistency. Our replica management system supports concurrent read/write and achieves week consistency with asynchronous replication. The method is similar to epidemic approach mentioned above. For details please read Section 3.1.

## 4.3 Data Transfer

GridFTP is widely used in the Gird for data transfer. There are two points in GridFTP that deserve a mention here. The first, GridFTP supports third party transfer, which is very important for replica management. Third party transfer is explained in the official Globus Alliance [42] like this, "The client, who will only orchestrate, but not actually take place in the data transfer, and two servers one of which will be sending data to the other". The second is the separation of front end and data node processes for security issues. The front end is responsible for client control connection. It can be run as any user, typically user `globus`, which has limited access to the machine. The back end is run as root and configured to only allow connections from the front end. As the back end runs as root, it can allow the server to `fork` and `setuid` on a child processes related to an authenticated user. If an attacker compromises the process, they could only obtain access to the limited account in the front end. In addition, single front end process may contact many back end data nodes. This is called *striped configuration*. It allows the combined bandwidth of all data nodes to be used.

RFT in GT4 is based on GridFTP. It complies with WSRF and functions as a "job scheduler". You can query the transfer status or subscribe for notifications of state change events after you provide a list of source and destination URLs to it.

In the thesis work, we only use GridFTP. However for better designs, we should use RFT to provide status-checking.

## 4.4 Security Issues

.In our system we take advantage of simpleCA [56] in the Globus toolkit 4 (GT4) [3] to set up Globus Grid Security Infrastructure (GSI) [5] for secret, tamper-proof, delegatable communication between services. In addition [4] provide a survey on decentralized security and the consequences of decentralized security among the Grid sites. In this paper, we will not pay many efforts on security issues.

## 4.5 Higher Level Replica Management

DRS [6] is a higher-level data management services for Grids. It uses the hierarchy and distributed RLS in the GT4 for replica location information, uses RFT and GridFTP for data transfer, and uses the security tools in GT4 for the security issues. There is no data consistency mechanism in DRS. It adopts WSRF to promote interoperability and represents the states of the replication request as resource properties. User initiates DRS by creating a request file that contains an explicit description of the replication request. The request may include file name, desired destination locations. Then resources for this request are created in the DRS. According to the request, the DRS checks replica position, creates new replicas and register its position in the LRC. The LRC will in turn update the information in all RLIs in the way mentioned in section 4.1. DRS lets user decide the destination position for the new replicas and uses a source selector class implemented by the user to select the desired source file.

Comparing with DRS, Replica Management Service (RMS) in the EU DataGrid [44] is more autonomic and more easily understood by the end-user. It adopts the OGSA concept for several components. The RLS in the RMS is almost the same as the RLS of GT. The difference is that the LRC stores the mappings between GUIDs (Grid Unique Identifiers) and PFNs instead of LFNs and PFNs. To maintain the mappings between LFNs and GUIDs, a Replica Metadata Catalog Service (RMC) is added. In addition the RMC provides metadata for the file represented by the GUID. With the metadata, the RMC gives users a way to query the file catalog based on attributes. RMS has two important APIs called *getNetworkCosts* and *getSECosts*. They monitor the network traffic and the access traffic to the storage device respectively. They interact with a service called Replica Optimization Service which can in turn perdict data access latencies. In EU DataGrid, optimization is performed in the three points, i.e. job scheduling, pre-execution and run-time [45]. The component called Resource Broker is responsible for the first point optimization. It decides which site has the least loaded resource with the maximum amount of locally available data for a job. After the job is allocated to a site and before it is executed, an optimization is performed to locate the best replicas for the files needed by the job based on an economic model. We will explain this model later. This kind of optimization is also performed during the job execution (run-time). For the security issue, RMS uses the EDG Java security package.

There are different replica selection algorithms for a file needed by job execution. In our system it is based on the “ant” and user QoS requirements. In the EU DataGrid, an economy-based file replication strategy is provided. In the model, data files are regarded as the goods in the market. Each site has an optimization agent. They are interacted with each other through auction protocol for optimal dynamic replica selection according to file requests from the job execution. A little more details go like this. Firstly a job execution site brings out the request or “calling a bid” for a file and this bid is propagated to the other Grid sites. For each site, once it receives the request, it first checks if it has the replica for this file. If so, it calculates a bid proportional to the transfer time between the calling site and itself and returns the result to the job execution site. The site that submits the lowest bid wins and the job execution site accesses the replica in the winner site. If the site doesn’t have the replica, it may start a nested auction according to a prediction function. The prediction function tells the site whether it is economically beneficial to create a new replica at local place according to the file access history. If a nested auction is initiated, the site that starts it also responds to the job execution site with its own bid.

Comparing with DRS, the RMS in the EU DataGrid and our system implement replica selection algorithm to locate appropriate data and relieves the burdens of the application programmers. The RMS provides the optimization, from job submission to replica selection, while our system focuses on the replica selection for the user site. The RMS selects replica according to the network traffic and the access traffic to the storage device. Our system allows user to provide their RTT requirement and only consider network characters. For the replica selection, the RMS creates the new replica only when it thinks this is beneficial in the long term. Our system creates the new one at any place that can reduce file access time. Consideration for future replica distribution is left to the Statistics

Service. Therefore our method is more flexible and may be better for reducing access time in each file access.

# 5 Prototype Implementations

In this thesis work, a prototype of replica management framework is implemented with Java and Globus Service Build Tools (GSBT) [46] in the GT4 environment, including replica selection component, statistics component, location information component and data consistency component. Some work takes advantages of DKS, FAM and several GT4 tools. In particular, the RLS is based on DKS; the data consistency component is based on the open source project, FAM (including *fam* daemon and a library in C language); the NLS is based on the GT4 default index service; the GridFTP client in the data transfer component use JavaCog to access GridFTP servers. Except this, all the other work mentioned in the following is implemented by this thesis. The rest of this chapter will describe these implementation details.

## 5.1 Data Consistency Component

The data consistency component consists of two layers, the lower FAM Java API and the upper application for data consistency maintenance. All are implemented by this thesis. Figure 6 depicts the Data Consistency component framework. FAM in essence includes the FAM daemon, *fam*, and a library for interacting with this daemon. The library provides the functions to open a connection to fam, monitor a file or directory, control the status of monitor and detect changes. These functions are implemented in C by an open source project [47]. To take advantage of the FAM C library and export a FAM Java API to the upper application, JNI is used. Implementing calls from Java to the FAM C library is a three-step process. The first is to write declarations in Java for the native method (C function). These declarations are in the 4 classes: the class "*FAM*" initiates the event constants and defines methods for opening a connection to the fam daemon; the class "*FAMRequest*" identifies a file or directory monitor and declares methods for controlling monitor status; the class "*FAMEvent*" declares methods related to the fam events, such as returning the events numbers or printing events; the class "*FAMConnection*" declares exact methods for monitoring a file or directory. The second step is to generate the header file for the next step. This header file, which is called "*fam-jni.h*", is created with the Java SDK's *javah* tool. The last step is to implement the native methods in C. These functions use header file in the previous step, make calls to the FAM C library and return results to Java.

The upper data consistency applications are built on top of the FAM Java API. They include the "*DirectoryMonitor*", "*FileMonitor*", and a tool class called "*RLSOP*". DirectoryMonitor monitors the storage element default directory "/home/icebox/filedir", but it can also monitor any other directories if required by the local replica management system. Its constructor registers all files in the monitored directory to RLS. Then it runs as a thread and listens to the FAM events. When a file within this directory is changed and the "changed" event happens, it will check the replica numbers of this file in the RLS. If the number is zero, which shows the file is never registered and is not cared by the replica management system, it is most likely to be produced temporarily by the local file system and the changes can be ignored. Therefore no further action is needed to do. If there is only one replica record in the RLS, nothing needs to do either, because no consistency maintaining work is required for only one replica. If there is more than one

replica, it will check whether this file is associated with a file monitor. Any file without a corresponding file monitor will cause a new thread (file monitor) to be created and started. We will talk about the file monitor in the next paragraph. When the "delete" event happens, which means a file within the monitored directory is removed, we check whether this file is monitored. If so, this file name is deleted from an ArrayList, called *fileMonitored*, which is used to record which files are associated with a FileMonitor thread. For other FAM events, such as "execution" and "create", the DirectoryMonitor thread just ignores them. The data consistency component doesn't consider the "create" event, because it can't discriminate the temperate files produced by local file system and the "true" new files that really matter to the replica management system, and therefore can't decide whether the file information should be added into the RLS. So we let our component pass "create" event away and not add new item (file-position pair) into the RLS. Instead it is left to the applications when they create a file. In particular, whenever the application creates a file which matters to the replica management system, it should add this file name and location information into the RLS.

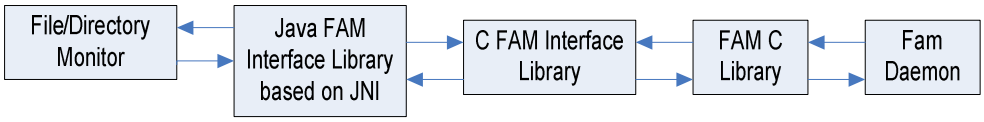


**Figure 7 Data Consistency Component Framework**

The class "FileMonitor" runs as a thread. Every monitored replica has a file monitor thread associated with it to maintain its consistency with other replicas. Whenever a FAM "change" event is received, its time is recorded. At every hour, the FileMonitor checks whether there is any file change within this time period. If there is, it copies the file to the other replica locations. If the file is deleted, the monitor thread updates the RLS and kills itself.

The class RLSOP serves as a tool box. It provides the API for add, query, delete and other operations related to the RLS. It also plays the role of synchronizing data replicas. Currently we copy the new replicas to the other places when required by the FileMonitor. This may cost too much time and bandwidth. In the future we can improve it by only sending the differences to the other replicas. In fact there is a way to produce differences between any two binary files, called XDelta [55].

## 5.2 Statistics Component

This component is a web service, responsible for proactively placing and deleting useless replicas. In general it is deployed in one Grid node. However to avoid centralization bottleneck and single-point failure, we can deploy it in a distributed and hierarchy way, just as how the index service in GT4 does.
The operations in the Statistics service include:

- *collectAccessInfo*. After a delegate agent gets replica selection results, it notifies the user to get results. Meanwhile it employs the "collectAccessInfo" to submit information related to the selection results to the Statistics component. The Data Consistency component does the same thing periodically to report file changes. The submitted information contains the file name and its position. In the current

implementation we use three HashMaps to store user access information. As an alternative, we are thinking to use the database and implement some statistics information as resource properties based on WSRF.

- *addClient*. This operation is used for proactively placing replicas. To require the statistics service to place popular files in advance, the user needs to register its position and QoS requirement in the Statistics component. This register operation is what addClient does. In addition inside the Statistics component, a HashMap works as a container for recording user QoS requirement.
- *removeClient*. This operation is the contrary to the addClient. It removes the register of the user position and requirement from the Statistics Component. Any proactive placing action will not be committed any more.
- *reset*. This operation clears all access records and user registers in the Statistics Component, such as previous replica selection result collections and the name set of users who commit the "addClient".

Statistics Service schedules a thread (class "*MyServices*") at a fixed rate whose current values are 7 days. "MyServices" is the core of Statistics service. It gets the file lists from each node and compares them with previous replica access records. The file which is not accessed for a long time is removed. Here we assume that the user employs replica selection component to pick up appropriate file position before he access any files. Therefore the replica selection records plus file change records reflect how files are accessed within a period in some sense. In addition, with these records, "MyServices" can determine which files are popular. According to the user requirements, these files are placed near the users. The methods "*deleteUseless*" and "*proactivePlacement*" are in charge of these two tasks mentioned above.

In the "*impl*" directory of Statistics component, there are two other classes, called "*ForAliveInfo*" and "*ValueListener*". They in essence work as two clients. The "ForAliveInfo" is the client for the NLS. It checks the NLS and gets the nodes position list. "ValueListerner" acts as the replica selection results listeners. When the MyServices acts as the delegate agent to proactively place replicas, this class is responsible to receive the results from ants and schedule replica movements.

# 5.3 Location Information Component

Location Information Component provides replica location information and Grid node location information within a virtual organization. It includes two parts, Replica Location Service and Node Location Component.

## 5.3.1 Replica Location Service

In the implementation, this service is called RLSDKS. It is built on top of the DKS middleware and exports many DKS APIs as remote operations. Whenever the GT4 web container starts up, the RLSDKS in it will contact the NLS to get the current Grid living node lists. Therefore there should be at least one working NLS before any RLSDKS starts up.

The RLSDKS initialization goes like this. Firstly it checks the NLS to find if there is any registered node. If it itself is the only one registered in the NLS, the RLSDKS will set up a new DKS ring and act as the first node. Otherwise the RLSDKS will contact any node in the list to get a DKS node URL as the entry point to join in the existing DKS ring. In this way, the DKS ring is constructed and includes all registered nodes.

The RLSDKS service contains the following operations.

- *add*. It receives the file name and its position as input parameters. In the DKS, the key is the "long" type. Therefore this operation takes the hash code of file name as the key and file position as the value. However there is a small bug here. Different file names may has the same hash codes. In the future we should consider other ways to guarantee the key is unique for each file.

- *delete*. This operation is almost the same as the add operation. The difference lies in the different DKS API it uses.

- *query*. The only parameter to this operation is file name. This operation queries the DKS ring to get all file positions for this file. Before it returns the file positions to the user, this operation checks the local cache which contains the node lists obtained from NLS. It compares the query results with the node lists. Any position which is not in the list is removed. In this way we hope to update RLS and maintain the consistency between the replica location information and physical file distribution.

- *getFirstDKSNodeURL*. It returns a DKS node URL for which other new Grid node may take as the entry point to the DKS ring.

- *getSuccessor*, which returns the successor IP for the current node in the DKS ring.

- *getRoutingTable*, which returns the IP addresses of the nodes which are in the first level (level 0) of local node DKS routing tables.

In addition, the RLSDKS schedules a thread at a fixed rate (the current rate is one minute). This thread is called *PeriodAction*. It is in charge of updating node lists cached locally by getting newest list from NLS.

## 5.3.2 Node Location Component

This component consists of NLS and "*aliveinfo*" service. The NLS is based on the GT4 default index service. In our system, there is only one node that has the NLS. It should be set up before any other sites start up. A configuration file in XML that specifies registrations to the NLS is created. This file is called "*alive-position-registration.xml*". Whenever a node joins a virtual organization, it should run "*mds-servicegroup-add*" to perform the registrations specified in this configuration file. Currently this is done manually. In the future it is better to integrate this registration process in the "aliveinfo" service. So whenever the "aliveinfo" is initiated, it automatically registers resources in the NLS.

Our NLC uses the *QueryAggregatorSource* to collect information from registered resources in the “aliveinfo” services by WS-Resource Properties polling mechanisms. Figure 6 describe this framework. In general the corresponding configuration file specifies a grid resource to be registered, a service group to be registered to, and various parameters associated with the registration. In our implementation, we register the resources properties in the "aliveinfo" service, i.e. the service address and the file list. We register them to a service group in the NLS. The registration is refreshed every 10 minutes and the resource properties are refreshed every 1 minute.

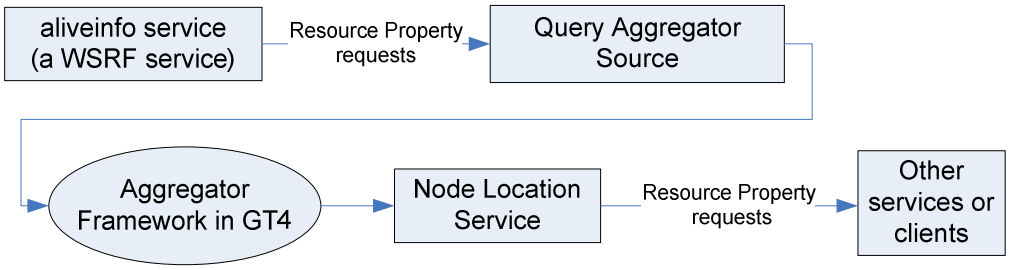


**Figure 8 Node Location Component Frameworks**

The "aliveinfo" is deployed at each site. It has two resource properties, IP address of local site, and file lists of local storage element. Although this service has only two resource properties now, it builds a good foundation for the further development. In the future, we can add more resources properties like local work load and storage response time to it. This could provide ants more information besides RTT to select replica location.

## 5.4 Replica Selection Component

This component includes two services, "*Myagent*" and "*Notification*". "Myagent" receives user requests, senses the network environment by computing RTT, and interacts with each other through the ants.

### 5.4.1 Agent Service

There are five classes in its "*impl*" directory. The class "*NotifClient*" provides the function to notify the user of selection results by modifying the resource properties of a specific notification service. It has two methods, "*add*" and "*printResourceProperties*". The "add" method changes the resource properties, while "printResourceProperties" shows the current status of replica selection. The interface "*NotifQNames*" defines several QName for the "Notifclient". The class "*ForAliveInfo*" is in essence a client to get the node location list from the NLS.

In addition the agent service employs a package called *roundtriptimer* to compute the RTT between local site and the user site. The package contains three classes, *PingClient*, *PingPong* and *PongManager*. The PingClient sends 10 TCP packages to a specific destination and compute the average round trip time. The PongManager is a server to receive the TCP packages. As the PongManager starts up, it constructs a thread list and initializes ten threads in the list which are free and ready to receive remote connection.

Then it listens at the port 4040. Whenever a connection request is received, the PongManager gets a free thread from the thread list to handle the communication between the request site and itself. This thread replies a TCP package whenever it receives one. The class "PingPong" is the implementation for this kind of thread. The “PingPong” also has a static request pool. The scheduled thread can take requests from this pool when it is motivated by the PongManager.

The core of agent service lies in the class "Myagent". It starts by getting PongManager work, which enable it to respond to any Ping requests. It has a static class "*DelegationStatus*". If a user presents a request to an agent, this class will record the data related to a replica selection corresponding to this user request. These data are the file name and RTT requests, received ant numbers and final results. In the current implementation, we use the static class to record the status, this may seems awkward. In the future, we may replace it with several resource properties, which can strength the interoperability of this service.

The following operations are provided in this service.

- *receiveRequest*. Input parameters to it are requested file name, RTT requests and user site address. This operation firstly checks its delegate status. Only when the status is free, it accepts user requests. Afterwards it gets the replica positions from local RLSDKS and computes their RTT from these positions. Then it determines whether any node can satisfy the requirement. If no node is all right, it commits the "motivate" operation in these replica positions. If the delegate status is busy, it will ignore this user request and reply to the user with busy information.

- *motivate*. It initiates ants with user requests and sends them or in other words conducts *antContainer* operation of the nodes that are in the first level of local node DKS routing table. A local method called "*sentAnt*" is used for sending ants. This method checks out whether the destination site is local node. If so, it will take the successor node of the destination site as the destination. This check process will continue until a non-local site is found.

- *antContainer*. This operation implements the ant algorithm. It receives ants, checks its status, and takes the corresponding action. If the hops in the status are beyond the *MAX_HOPS*, the ant container will send this ant back to the delegate agent which in turn will notify the user. Otherwise, it starts the PingClient to get the RTT and compares it with the ant status. The ant status is updated correspondingly.

- *getFileSize*. This operation returns the size of a specific file if this file exists in the local storage element.

- *getRTT*. This operation computes the RTT from the local site to a remote node.

- *getFileLists*. This operation gets all files names from the local storage element.

### 5.4.2 Notification Service

This service is simple. It contains a resource property called "*informationRP*", which is just a string chaining the user site, RTT requirement, minimum RTT and its position. We didn't put this information into four separate properties. This is to avoid possible information mismatch when several add operations are conducted at the same time. By bundling them into a single string, we are sure about the matches between these four parameters. In addition we declare this property as a topic, so the user site can subscribe to it and receive the notification when it is changed. This service has only one operation, called add. It can update the resource properties with new values.

## 5.5 The Components and The Corresponding Classes

For the convenient of future work, we indicate the relation between the components and the classes in this section.

| Component Name | Services Name | Class Name |
|---|---|---|
| Replica Selection | Agent | AntModel, CacheWorker, ForAliveInfo, Myagent, MyagentQNames, NotifClient, NotifQNames, WorkerForAntContainer, WorkerForSendingAnt |
| | Notification | NotifClient, NotifQnames, NotifService |
| NLC | RLSDKS | ForAliveInfo, PeriodAction, RLSDKS |
| | AliveInfo | AliveInfo, AliveInfoQNames, AliveInfoResource, AliveInfoResourceHome |
| Statistics | statistics | ForAliveInfo, MyServices, Statistics, ValueListener |
| Data Consistency | | DirectoryMonitor, FileMonitor, RLSOP, CogGSIUtils, GeneralUtils, GridFTP, ProxyProducer |

# 6 Profiling of Prototype

This section presents performance evaluation for our implementation. The first part is the time anatomy of replica selection. The complete process of replica selection is divided into several phases. We expect to observe the roles which different phases play in the time consumption.

Up to now, we don't have an appropriate test-bed platform to deploy more performance evaluation. But more tests and experiments could be performed, if the lab environment is available. So at the end of this chapter, we propose several measurements which may be achieved in the future.

## 6.1 Time Anatomy of Replica Selection

In this evaluation, one site, the user, submits its replica selection requests to the other, a Grid site that runs a GT4 container as well as Replica Selection service, Notification service and the three lower-level components. The user site is Intel Pentium processor 1.6 GHz, 512MB RAM and Intel 802.11b wireless card. It sends the replica requests for a file whose RTT requirement is 100 milliseconds and local storage space it could provide is zero, i.e. the file is not allowed to be placed in the user site. The size is set zero, so we have the chance to observe the time consumption of complete replica selection process. In addition the Grid site is .

Before the test, we put the requested file in the Grid site in advance, so there is at least one replica location record for this file in the RLS. The RTT delays for replica selection between the two sites are generated randomly. In particular, when one site needs to measure the RTT between itself and the other, it sends out a TCP packets to the other. The other site, as it receives it, will reply after a random delay. The delay time is uniformly distributed value between 0 and 1000 milliseconds.

We divide the complete replica selection process into the following phases and report the time consumption for them.

- *Submit request* is the phase for the user to create a resource in notification service and subscribe to its properties for result notification.
- *Query RLS* is the phase for the delegation site to query the RLS to get the replica position.
- *Judge the existing replica site* includes two steps. The first is to get the file size from one replica position that has the file and decide whether the user site has the storage space to hold the file. The second is to decide whether the existing replica site can satisfy the RTT requirement.
- *Seeking appropriate positions by motivating ants* is the phase to use the ant walking around to pick an appropriate position.
- *Register the new replica* is the phase to copy the replica to the new site and register its position in the RLS. In our performance evaluation, the copy time to create new replica is ignored, for this time depends on actual file size and

GridFTP configuration which may vary greatly between different sites and can not reflect our system performance.

- *Inform the user* is the phase to change the resource properties in the notification service with the final results which in turn notifies the user site.

In "judge the existing replica site" phase, we "force" the Grid site that already has the file generates longer delay than the RTT requirement, i.e. it will not meet the RTT requirement in this step. So the replica selection process can go on further and we are able to observe the complete process of replica selection. In addition, due to the fact that we only have one Grid site whose DKS routing table only contains itself, the ant, in fact, will just circle around in one site. So the final replica selection result will be either this Grid site whose random delay is less than 100 sometimes or no satisfying site at all. Finally the *k* parameter of the DKS in our measurement is two, i.e. the first level of routing table has two items and two ants are motivated. We set ant walking step as 5. We submitted the request five times and obtained the average performance shown in Table 2.

**Table 2 Time Anatomy of Replica Selection**

| Phase | Time(ms) | Standard Deviation |
|---|---|---|
| (1) Submit request | 426 | 32 |
| (2) Query RLS | 144 | 15 |
| (3) Judge the existing replica site | 618 | 24 |
| (4) Seeking appropriate positions by motivating ants | 1155 | 957 |
| (5) Register the new replica in the RLS | 271 | 19 |
| (6) Inform the user | 615 | 20 |

We observe the variance is large on the phase of "seeking appropriate positions by motivating ants". This is due to the fact that our delay for measuring RTT is generated randomly and the ant could encounter different RTT as it walks around. In the practice WAN environment where network traffic may change greatly, the variance in this phase will also be large. Additionally the time in the phase of "query RLS" is not so large. As the RLS has only one item, this may be reasonable. But the time in this phase depends on numbers of items stored in the RLS greatly. As more location information is stored in the RLS, this query time will increase.

The Figure 9 shows the time consumption distribution of different phrases. With no surprise, the phrase 3 and 4 rank the top 2. Both of them need to cooperate with other sites quite often to get information, which is time-consuming. Therefore how to reduce the needs to communicate with services in other sites is the key to decrease replica selection time. The phrase 6 and phrase 1 also play important roles in time consumption. They are both related to the operations on the web service resources, which may account for the large time consumption.

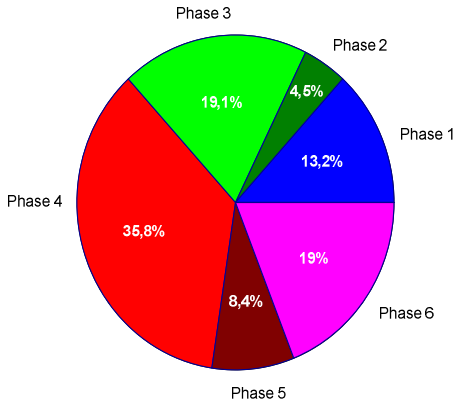


**Figure 9 Time consumption distribution for replica selection**

# 6.2 The Future Work in Performance Evaluation

It is very interesting to observe how the system performs under different replica selection policies. So we could do the following two experiments in the future.

**Ants Numbers and Their Walking Steps**

We will see how the time and RTT rate are sensitive to the number of ants and their walking steps. Here "RTT rate" refers to the rate between the results RTT and RTT requirements. For example, the user proposes the RTT requirement as 200ms. Our replica selection method gets a position with RTT 180ms. So the RTT rate is 90%. Obviously we hope the RTT rate is as small as possible. The "time" refers to the replica selection time and doesn't include the file transfer time. The user site will access 100 files or more. We will record the time consumption for each replica selection.

**Cache Effects**

We will see how the time and RTT rate are sensitive to the caches existence. Here "RTT rate" refers to the rate between the results RTT and RTT requirements. For example, the user proposes the RTT requirement as 200ms. Our replica selection method gets a position with RTT 180ms. So the RTT rate is 90%. Obviously we hope the RTT rate is as small as possible. The "time" refers to the replica selection time and doesn't include the file transfer time. The user site will access 100 files or more. We will record the time consumption for each replica selection.

**File Access Patterns**

We will observe the performance of our replica selection method under different file access patterns. Firstly each file name is mapped to a unique integral. Then (a) the user selects 100 integrals randomly which meets the uniform distribution to access 100 files; (b) the user selects integrals for 100 times which conform to the Gaussian distribution to access files.

# 7 Conclusions and Future Work

The Grid computing doesn't only belong to scientific fields that have limited numbers of processing and data storage nodes. It is moving towards enterprises applications and may integrate millions or even billions of IT resources. So when we design and deploy our Grid system, one of our concerns is its scalability and how easily we manage large number of resources within it. This thesis work focuses on managing file-based replicas among Grid nodes. We take the scalability into consideration with the help of P2P technology and ant algorithm. In addition by compliance with the spirits of autonomous computing, we expect to make the management work a little easier. As a plus, our system can select replicas according to the user specific QoS requirement. In particular, we layer our design into low level and high level and implement them within GT4 environment. The low level includes Node Location Component, Data Transfer Component, and Data Consistency components. NLC integrates the DKS in the RLS and builds Node Location Service on the GT4 WS MDS Aggregator Framework to provide replica location information. Data transfer component is based on GridFTP to create or move replicas. The Data consistency component maintains the consistency with the help of FAM. The high level takes advantage of low one and includes two components: replica selection component and statistics component. While the former provides service of selecting replica according to user's specific QoS requirements, the latter proactively places replicas according to the user access records.

As far as the performance of our system is concerned, we pay more attention to the system response time and result precision to the RTT requirement in replica selection component. Different design polices, including the ants numbers, ant walking steps and the introduction of stigmergy mechanism (or cache), have effects on the performance. Additionally file access patterns also play a role. We argue that different application cases should consider these factors to improve system performances.

To reduce file access time of user node, we pick up or create a replica on the node that has a required RTT to the user node. In our implementation, the RTT is estimated based on TCP protocol. However using RTT for predicting access time is not good enough. It is far from being precise. Firstly the user node may use different patterns and protocols to access files. Secondly the file access time is affected by the storage component of remote node too. Some factors, such as workloads of the remote node and storage device types may influence file access time. However these factors are not considered. In our future work we plan to take more factors into consideration, such as storage response time. Although current method needs to be refined, it does reflect network dynamism in some sense and gives us a prototype and guide on the design choices.

The current Statistics component in the system collects user access records and proactively creates replicas for the files which have been accessed most often. This is based on the assumption that popular files are highly possible to be accessed in the future. By putting them near the user node, we make preparation for their future use. As you can see, this method predicts future file access based on previous file access times. More advanced prediction method can be developed in future work. We plan to predict file

access according to the application type, such as high energy physics application, biomedical applications, and earth observation science application. We believe different application type has different file preferences. According to the application type and previous access records, we expect to improve the success rate of prediction.

In addition, we plan to take advantage of more P2P overlay network characters for the replica management. For example, some P2P overlay networks are topology-aware; using their knowledge relating to distances between the nodes may help ants to find appropriate positions more quickly.

# 8 Lists of Abbreviations

| | |
|---|---|
| CAS | - Complex Adaptive System |
| DHT | - Distributed Hash Table |
| DKS | - Distributed K-ary System |
| DRS | - Data Replica Service |
| FAM | - File Alteration Monitor |
| GRAM | - Globus Resource Allocation Manager |
| GT | - Globus Toolkits |
| GT3 | - Globus Toolkit, version 3 |
| GT4 | - Globus Toolkit, version 4 |
| GSBT | - Globus Service Build Tools |
| GSI | - Grid Security Infrastructure |
| JNI | - Java Native Interface |
| LFN | - Logical File Name |
| LRC | - Local Replica Catalog |
| MDS | - Management and Discovery Service |
| NLS | - Node Location Service |
| NLC | - Node Location Component |
| OGSI | - Open Grid Service Infrastructure |
| PFN | - Physical File Name |
| P2P | - Peer-to-Peer |
| QoS | - Quality of Service |
| RFT | - Replica File Transfer |
| RLI | - Replica Location Indices |
| RLS | - Replica Location Service |
| RMC | - Replica Metadata Catalog Service |
| RMS | - Replica Management System |
| RTT | - Round Trip Time |
| RS | - Replica Selection |
| TTL | - Time To Live |
| WSRF | - Web Service Resource Framework |
| VO | - Virtual Organization |

# 9 References


[1]. I. Foster. What is the Grid? A three point checklist. *Grid Today*, July 20, 2002.

[2]. D.Dullmann, W.Hoschek, J. Jaen-Martinez, B. Segal. Models for replica synchronization and consistency in a data Grid. In *10th IEEE international symposium on high performance distributed computing.*

[3]. Globus Toolkit 4. URL: http://www.globus.org/toolkit/

[4]. D. Thain, C. Moretti, P. Madrid. The consequences of decentralized security in a cooperative storage system. In *IEEE workshop on security in storage*, San Francisco, December 2005.

[5]. I. Fonster, C. Kessleman, G. Tsudik and S. Tuecke. A security architecture for computational grids. In *ACM Conference on Computer and Communications Security Conference*, 1998.

[6]. A. Chervenak, R. Schuler, C. Kesselman, S. Koranda, B. Moe. Wide Area Data Replication for Scientific Collaborations. *Proceedings of 6th IEEE/ACM International Workshop on Grid Computing (Grid2005)*, November 2005

[7]. W.E. Allcock, I. Foster, R. Madduri. Reliable Data Transport: A Critical Service for the Grid. *Building Service Based Grids Workshop, Global Grid Forum 11*, June 2004.

[8]. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. *IFIP International Conference on Network and Parallel Computing*, Springer-Verlag LNCS 3779, pp 2-13, 2005

[9]. GridFTP Protocol Specification (Global Grid Forum Recommendation GFD.20). March 2003

[10]. M. Baker. Ian Foster on Recent Changes in the Grid Community. *IEEE Distributed Systems Online Vol.5, No.2; Feb, 2004.*

[11]. M.M.Waldrop. Autonomic Computing, The Technology of Self-Management. The Future of Computing Project at the Woodrow Wilson International Center of Scholars, Dec 2003.

[12]. J.Crowcroft, T. Moreton, I. Pratt, A. Twigg. Peer-to-Peer systems and the Grid. http://www.cl.cam.ac.uk/users/jac22/out/.

[13]. M.Cai, A. Chervenak, M.Frank. A Peer-to-Peer Location Service Based on A Distributed Hash Table. *Proceedings of the SC2004 Conference (SC2004)*, November 2004.

[14]. A. Chazapis, A. Zissimos, N. Koziris. A peer-to-peer management service for high-throughput Grids. *Proceeding of the 2005 International Conference on Parallel Processing.*

[15]. W. Acosta, S. Chandra. Unstructured Peer-to-Peer Networks-Next Generation of Performance and Reliability. *Refereed poster, IEEE INFOCOM 2005.*

[16]. E. Cohen and S. Shenker. Replication Strategies in Unstructured Peer-to-Peer networks. In Processings of the 2002 conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, page 177-190. ACM Press, 2002.

[17]. S.C.Rhea and J. Kubiatowicz. Probilistic location and routing. In Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2002), June 2002.

[18]. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM' 01 Conference*, San Diego, California, August 2001.

[19]. B.Y.Zhao, J.D. Kubiatowicz, and A.D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, U.C.Berkeley, April 2001.

[20]. A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *International Conference on Distributed Systems Platforms (Middleware)*, Nov. 2001.

[21]. Nicholas J.A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, Alec Wolman, SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS)*, Seattle, WA, USA, March, 2003.

[22]. Ali Ghodsi, Luc Onana Alima, Seif Haridi. Symmetric Replication for Structured Peer-to-Peer Systems. In *The 3rd International Workshop on Databases, Information Systems and Peer-to-Peer Computing*, August 28-29, 2005, Trondheim, Norway

[23]. Ali Ghodsi, Luc Onana Alima, Seif Haridi. Low-Bandwidth Topology Maintenance for Robustness in Structured Overlay Networks, In *the 38th International HICSS Conference*, Springer-Verlag, January, 2005.

[24]. Luc Onana Alima, Ali Ghodsi, Per Brand, Seif Haridi. Multicast in DKS(N, k, f) Overlay Networks, In *Proceedings of the 7th International Conference on Principles of Distributed Systems*, Springer-Verlag, Berlin, 2004

[25]. Ali Ghodsi, Luc Onana Alima, Sameh el-Ansary, Per Brand, Seif Haridi. Self-Correcting Broadcast in Distributed Hash Tables. In *Series on Parallel and Distributed Computing and Systems* , ACTA Press, Calgary, 2003

[26]. Sameh El-Ansary, Luc Onana Alima, Per Brand and Seif Haridi, Efficient Broadcast in Structured P2P Networks. In *the 2nd International Workshop On Peer-To-Peer Systems*, (Berkeley, CA, USA), February 2003

[27]. I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. *IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779, pp 2-13, 2005*

[28]. A. Chervenak, R. Schuler, C. Kesselman, S. Koranda, B. Moe. Wide Area Data Replication for Scientific Collaborations. *Proceedings of 6th IEEE/ACM International Workshop on Grid Computing (Grid2005), November 2005*

[29]. A. Chervenak, E. Deelman, I. Foster, L. Guy, W. Hoschek, A. Iamnitchi, C. Kesselman, P. Kunst, M. Ripeanu, B, Schwartzkopf, H, Stockinger, K. Stockinger, B. Tierney. Giggle: A Framework for Constructing Scalable Replica Location Servcies. Proceedings of Supercomputing 2002 (SC2002), November 2002.

[30]. Hawkeye. URL: http://www.cs.wisc.edu/condor/hawkeye/.

[31]. Ganglia. URL: http://ganglia.sourceforge.net.

[32]. MDS Key concepts. URL:http://www.globus.org/toolkit/docs/4.0/info/key-index.html

[33]. Wei Lu, Kenneth Chiu, Aleksander Slominski, and Dennis Gannon. "A streaming validation model for soap digital signature". In *14th IEEE International Symposium on High Performance Distributed Computing (HPDC-14)*, 2005

[34]. The Globus Security Team. http://www.globus.org/toolkit/docs/4.0/security/GT4-GSI-Overview.pdf. GT4 Grid Security Infrastructure: A Standard Perspective.

[35]. J. O. Kephart, and D. M. Chess. "The vision of autonomic computing". IEEE Computer 36(1):41-50, 2003

[36]. Steve Graham, Doug Davis, etc. Building web services with Java (Second Edition), 383, 2005 by Sams Publishing.

[37]. Mark Baker. "Ian Foster on Recent Changes in the Grid Community". *IEEE Distributed Systems Online*, February 2004.

[38]. M. Cai, A. Chervenak, M. Frank. A Peer-to-Peer Replica Location Service Based on A Distributed Hash Table. *Proceedings of the SC2004 Conference (SC2004),* November 2004.

[39]. H. Stockinger, Distributed Database Management Systems and the Data Grid, In *18th IEEE Symposium on Mass Storage Systems and 9th NASA Goddard Conference on Mass Storage Systems and Technologies*, San Diego, April 17-20, 2001**.**.

[40]. Versant, Inc. http://www.versant.com

[41]. ObjectStore http://www.exceloncorp.com/products/objectstore.html

[42]. Third-party-transfers http://www.globus.org/toolkit/docs/4.0/data/gridftp/GridFTP_Glossary.html#third-party-transfers

[43]. LIGO Project. Lightweight Data Replicator, http://www.lsc-group.phy.uwm.edu/LDR/, 2004

[44]. P. Kunszt, E. Laure, H. Stockinger, and K. Stockinger. File-based Replica Management, *Future Generation Computer Systems*, 22(1):115-123, 2005, Elsevier

[45]. W. H. Bell, D. G. Cameron, L. Capozza, P. Millar, K. Stockinger, and F. Zini. Design of a Replica Optimisation Framework. Technical Report *DataGrid-02-TED-021215, CERN, Geneva, Switzerland*, December 2002.

[46]. Globus Service Build Tools. http://gsbt.sourceforge.net

[47]. FAM. http://oss.sgi.com/projects/fam

[48]. JNI http://java.sun.com/j2se/1.5.0/docs/guide/jni/index.html

[49]. M. Resnick. Turtles, Termites, and Traffic Jams: Exploration in Massively Parallel Microworlds. *MIT Press*, 1994.

[50]. Junwei Cao. Self-Organizing Agents for Grid Load Balancing. *Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)* pp. 388-395

[51]. A. Montresor, H. Meling, and Ö. Babaoglu. Messor: Load-Balancing through a Swarm of Autonomous Agents. In Proc. Of 1st Int. Workshop on Agents and Peer-to-Peer Computing, Linköping, Sweden, pp. 81-89, 2002.

[52]. L. Guy, P. Kunszt, E. Laure, H. Stockinger, and K. Stockinger. Replica Management in Data Grids. *Technical Report, GGF5 Working Draft, July 2002.*

[53]. Gnutella. http://rfc-gnutella.sourceforge.net

[54]. P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. *1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, March 2002.
[55]. Joshua P. MacDonald, XDelta, http://www.XCF.Berkeley.edu/~jmacd/xdelta.html
[56]. SimpleCA. http://www.globus.org/toolkit/docs/4.0/security/simpleca/
[57]. Complex Adaptive System. http://www.cs.iastate.edu/~honavar/alife.isu.html
[58]. K. Czajkowski, D. Ferguson, I. Fonster, etc. The WS-Resource Framework, version 1.0. http://www.globus.org/wsrf/specs/ws-wsrf.pdf

# 10 Appendixes

## Appendix A Java Doc

This appendix contains the Java doc. To save the file spaces, it doesn't include the documentary for all the Java class.

- Directory Monitor and File Monitor

### *Class DirectoryMonitor*

public class **DirectoryMonitor**
implements Runnable

This class monitor a specific directory.

## Constructor Detail

#### DirectoryMonitor

`public DirectoryMonitor(String path)`

If the directroy doesn't exist, exception will be thrown.

**Parameters:**

`path` - The directory to be monitored

---

#### DirectoryMonitor

`public DirectoryMonitor()`

The default monitored directory is "/home/icebox/filedir"

**Throws:**

`IOException` -

---

## Method Detail

#### initRLS

`public void initRLS()`

initRLS is used to load the local file list to the RLS.

---

#### start

`public void start()`

Open a connection to the fam daemon. Start the watch thread.

---

#### stop

`public synchronized void stop()`

Stop the monitor thread and close the connection to the fam daemon.

---

### run

```
public void run()
```

Capture the events from the fam daemon and do corresponding action.

---

### main

```
public static void main(String[] args)
```

---

## *Class FileMonitor*

public class **FileMonitor**
implements Runnable

FileMonitor Monitor a specific file.

# Constructor Detail

### FileMonitor

```
public FileMonitor(String path)
```

Initialize the timer and RLS operation.

**Parameters:**

`path` - The file to be monitored

---

# Method Detail

### start

```
public void start()
```

Open a conncetion to the FAM daemon and start up a watch thread.

---

### stop

```
public synchronized void stop()
```

Close the connection and kill the watch thread.

---

### run

```
public void run()
```

Capture the events and take corresponding action

---

### main

```
public static void main(String[] args)
```

---

*util*

## *Class RLSOP*

public class **RLSOP**

Description: This is a tool class. It provides API to access RLS and maintain the data consistency.

# Constructor Detail

### RLSOP

`public RLSOP()`

Get the RLS PortTypePort and initialization.

---

# Method Detail

### add

`public void add(String[] fileList)`

Before the file and its position are inserted into the RLS, this method will make sure there are not already existed

**Parameters:**

`fileList` - String[] The files name whose positions are to be inserted into the RLS.

---

### subtract

`public void subtract(String fileName)`

Delete a file position in the RLS

**Parameters:**

`fileName` - String The file name to be deleted from the RLS.

---

### existNum

`public int existNum(String fileName)`

Get the replica numbers for a file

**Parameters:**

`fileName` - String File name to be queried.

**Returns:**

int The number of replicas for a file

---

### positionExist

`public boolean positionExist(String[] positions)`

Check if the local position is in a position lists.

**Parameters:**

`positions` - String[] The positions list to be checked.

**Returns:**

boolean

---

### sync

```
public void sync(String fileName,
                 GSSCredential cred)
```

Data consistency Maintaining Note: although the security parameters are used here. But it is never tested. All the tests are performed without security consideraitons.

**Parameters:**

`fileName` - String The file name whose replica consistency is going to be maintained.

`cred` - GSSCredential Credentail for the security GridFTP access.

---

### getContainerUrl

```
public String getContainerUrl()
```

Get the local web container URL

**Returns:**

String

---

### getmyIP

```
public String getmyIP()
```

Get the local IP address

**Returns:**

String

---

- Statistics Service

***org.globus.myagent.services.core.statistics.impl***

## *Class Statistics*

public class **Statistics**

Description: This class collects user access information. It removes useless replicas and proactively places replicas

# Constructor Detail

### Statistics

```
public Statistics()
```

"MyServices" will be scheduled periodically.

---

# Method Detail

### collectAccessInfo

`public CollectAccessInfoResponse`
**`collectAccessInfo`**`(CollectAccessInfo params)`

Remotely-accessible operations collectAccessInfo. It collects file access info.

**Parameters:**

`CollectAccessInfo` - File name and file positions

**Returns:**

A empty stub

---

### addClient

`public AddClientResponse` **`addClient`**`(AddClient params)`

Remotely-accessible operation. The client submits it requirements.

**Parameters:**

`params` - AddClient client position and client RTTreq

**Returns:**

AddClientResponse A empty stub

---

### removeClient

`public RemoveClientResponse` **`removeClient`**`(String clientPosition)`

Remotely-accessible operation. The client removes its requirement for placing replicas around it.

**Parameters:**

`clientPosition` - String client IP address

**Returns:**

RemoveClientResponse A empty stub

---

### reset

`public ResetResponse` **`reset`**`(Reset param)`

Remotely-accessible operation. Clear all acess record

**Parameters:**

`param` - Reset A empty stub

**Returns:**

ResetResponse A empty stub

---

***org.globus.myagent.services.core.statistics.impl***

## *Class MyServices*

public final class **MyServices**
extends TimerTask

Description: This class is scheduled by the Statistics service periodically. It deletes useless and places replicas.

# Method Detail

### run

```
public void run()
```

Thread running.

---

### MaxForaccessInfo1

```
private String MaxForaccessInfo1()
```

**Returns:**

String The file name that has the most access time

---

### getFileList

```
private String[] getFileList(String position)
```

**Parameters:**

`position` - String Storage element address

**Returns:**

String[] File names

---

### deleteUseless

```
private void deleteUseless(String[] fileList,
                           String position)
```

**Parameters:**

`fileList` - String[] Files to be checked

`position` - String Positiions where files exist

---

### proactivePlacement

```
private void proactivePlacement(String fileName)
```

**Parameters:**

`fileName` - String Files to be placed.

---

### motivateNode

```
private void motivateNode(String filePosition,
                          String clientPosition,
                          long rttReq)
```

**Parameters:**

`filePosition` - String

`clientPosition` - String

`rttReq` - long

*org.globus.myagent.services.core.statistics.impl*

## *Class ValueListener*

public class **ValueListener**
extends Thread
implements NotifyCallback

Description: To receive notification from the reomote Notification service.

# Constructor Detail

### ValueListener

```
public ValueListener(String serviceURI,
                     String clientPosition,
                     long minRTTReq)
```

**Parameters:**

`serviceURI` - String The remote notification service address
`clientPosition` - String The user site address
`minRTTReq` - long The user RTT requirements

# Method Detail

### deliver

```
public void deliver(List topicPath,
                    EndpointReferenceType producer,
                    Object message)
```

This method is called when a notification is delivered

### run

```
public void run()
```

Thread running. To subscribe to the remote topic.

### getPosition

```
public String getPosition()
```

Get the final result: replica position

**Returns:**

String

### getValue

```
public long getValue()
```

Get the final result: min RTT value

**Returns:**
long

---

*org.globus.myagent.services.core.statistics.impl*
## *Class ForAliveInfo*

public class **ForAliveInfo**
extends BaseClient

Description: This class gets the current registered alive node sites addresses from NLS.

# Method Detail

### getPositionList

`String[] getPositionList()`
Get current alive registered node address.
**Returns:**
String[]

---

- Replica location Service

*org.globus.myagent.services.core.RLSDKS.impl*
## *Class RLSDKS*

public class **RLSDKS**

Description: This service registers in the DKS ring when it starts up and exports many RLS operations.

# Method Detail

### add

`public AddResponse add(Add params)`
Remotely-accessible operations. Add a replica position into the RLS
**Parameters:**
`params` - Add File name and its position
**Returns:**
AddResponse An empty stub
**Throws:**
`RemoteException` -

---

### delete

`public DeleteResponse delete(Delete params)`

Remotely-accessible operations. Delete a replica from the RLS

**Parameters:**

`params` - Delete File name and its position.

**Returns:**

DeleteResponse An empty stub.

**Throws:**

`RemoteException` -

---

### query

```
public QueryResponse query(String fileName)
```

Remotely-accessible operations. Query the positions for a file.

**Parameters:**

`fileName` - String

**Returns:**

QueryResponse File positions.

**Throws:**

`RemoteException` -

---

### getFirstDKSNodeURL

```
public String getFirstDKSNodeURL(GetFirstDKSNodeURL param)
```

Remotely-accessible operations. Get a DKS Node URL

**Parameters:**

`param` - GetFirstDKSNodeURL A empty stub.

**Returns:**

String URL for a DKS node.

---

### updateCache

```
public static synchronized void updateCache()
```

Update the local node list cache by contacting the NLS.

---

### getSuccessor

```
public String getSuccessor(GetSuccessor param)
```

Get the successor IP of the local site.

**Parameters:**

`param` - GetSuccessor

**Returns:**

String

---

### getRoutingTable

```
public GetRoutingTableResponse getRoutingTable(GetRoutingTable param)
```

Get the nodes in the level 0 of the local node routing table.

**Parameters:**
`param` - GetRoutingTable
**Returns:**
GetRoutingTableResponse

---

***org.globus.myagent.services.core.RLSDKS.impl***
## *Class PeriodAction*
public class **PeriodAction**
extends TimerTask

Description: This class is scheduled periodically to update the local cache.

# Method Detail

### run
`public void run()`
Update the cache.

---

- Agent service

***org.globus.myagent.services.core.agent.impl***
## *Class Myagent*
public class **Myagent**

Description: This service handles the ants and sense network dynamical environment by computing RTT

# Constructor Detail

### Myagent
`public Myagent()`
Constructor. Initialization. Start up the PongManager and clear the status

---

# Method Detail

### receiveRequest
`public RespondToRequest receiveRequest(ReceiveRequest params)`
To receive user requests and act as a delegate agent.
**Parameters:**
`params` - ReceiveRequest includes user IP, file name and RTT requirements.
**Returns:**

RespondToRequest contains the status of current agent.

**Throws:**

`RemoteException` -

---

### motivate

```
public MotivateResponse motivate(Motivate params)
```

Send out ants to the nodes in the first level of routing table.

**Parameters:**

`params` - Motivate includes user IP address, delegate agent address and user requirements.

**Returns:**

MotivateResponse An emptty stub.

**Throws:**

`RemoteException` -

---

### sendAnt

```
public void sendAnt(AntModel am,
                    String dest)
```

Check the vadility of the destination and send out ants.

**Parameters:**

`am` - AntModel

`dest` - String Destination IP address.

**Throws:**

`RemoteException` -

---

### antContainer

```
public AntAck antContainer(Ant ant)
```

The ant algorithm implementation

**Parameters:**

`ant` - Ant contains the status and data for ant algorithm

**Returns:**

AntAck An empty stub

**Throws:**

`RemoteException` -

---

### getSuccessor

```
public String getSuccessor(String param)
```

Get the successor IP of a node.

**Parameters:**

`param` - String node IP

**Returns:**

String Successor IP

---

### getRoutingTable

`public String[] getRoutingTable(String param)`

Get the routing table of a node

**Parameters:**

`param` - String node address

**Returns:**

String[] routing table of this node.

---

### getFileLists

`public String getFileLists(GetFileLists param)`

Get the files names of local storage element.

**Parameters:**

`param` - GetFileLists

**Returns:**

String File lists

**Throws:**

`RemoteException` -

---

### getFileSize

`public long getFileSize(String fileName)`

Get the size for a specific file in the local storage element.

**Parameters:**

`fileName` - String

**Returns:**

long The size of a file

---

### getRTT

`public long getRTT(String remoteClientPosition)`

Get the RTT between the local site and the remote place

**Parameters:**

`remoteClientPosition` - String The IP address of the remote place.

**Returns:**

long The RTT value.

---

***org.globus.myagent.services.core.agent.impl***

## *Class NotifClient*

public class **NotifClient**

Description: This class is used to notify the user site of final replica selection results. It changes the resource properties in the notification service.

# Method Detail

### add

```
public void add(String cp,
                long rttReq,
                String minRTTPosition,
                long minRTTValue)
```

Conduct the remote operation "add" in the notification service.

**Parameters:**

`cp` - String user site IP address

`rttReq` - long user site RTT requirement

`minRTTPosition` - String the selected position with the minimum RTT

`minRTTValue` - long The minimum RTT

**Throws:**

`Exception` -

---

### printResourceProperties

```
private static void printResourceProperties(NotificationPortType notif)
```

Query the resource properties in the remote notification service and print them

**Parameters:**

`notif` - NotificationPortType

**Throws:**

`Exception` -

---

- Notification service

***org.globus.myagent.services.core.notification.impl***

## *Class NotifService*

public class **NotifService**
implements Resource, ResourceProperties, TopicListAccessor

Description: This service takes the replica selection information as the resource properties. It adds this property as a topic. Therefore whenever it is changed, any one who subscribe it can be notified.

# Constructor Detail

### NotifService

```
public NotifService()
```

Constructor. Initializes RPs and topic

---

# Method Detail

### add

`public AddResponse add(Add params)`

Remotely-accessible operations. Update the resource properties.

**Parameters:**

`params` - Add includes the user IP address, RTT requirement and selected replica position and its RTT.

**Returns:**

AddResponse An empty stub

**Throws:**

`RemoteException` -

---

### getResourcePropertySet

`public ResourcePropertySet getResourcePropertySet()`

Required by interface ResourceProperties

---

### getTopicList

`public TopicList getTopicList()`

Required by interface TopicListAccessor

---

# Appendix B Use Cases

This appendix includes several use cases. They are: (1) Grid node setting up; (2) Replica Selection.

- Grid node setting up

1. The Grid node holding the Node Location Service starts up the web container. This web container shouldn't employ the RLSDKS.
2. A Grid node starts up the web container that could include all the components and services.
3. This Grid node runs "*mds-servicegroup-add*" with the configuration file "*alive-position-registration.xml"*. There is a parameter in it, corresponding to the NLS IP address. It should be set correctly.
4. Other Grid nodes are set up in the same way as 2 and 3.

- Replica Selection

1. The user creates a resource in a notification service and subscribes to its property, "informationRP".
2. The user submits replica selection requests including three parameters to any agent service. The parameters are file name, RTT requirement, local available storage space size. This agent is the delegate agent of the user.
3. The delegate agent queries local RLS about the existing replica location for the file.
4. The delegate agent queries a replica location to get the file size.
5. The delegate agent collects RTTs from all the existing replica locations.
6. The delegate agent motivates the existing replica locations which in turn send out ants.
7. Ants collect information and walk around the DKS ring. After fixed steps, it goes back to the delegate agent.
8. The delegate agent selects the potential position and computes the RTT between it and each existing replica position. It then moves the file from a position that has the least RTT to the potential position.
9. The delegate agent registers this position in the RLS and conducts the remote operation "add" in the notification service.
10. The delegate agent sends the results to the statistics service.
11. The notification service notifies the selection result to the user.

# Appendix C WSDL Files

This section contains the WSDL files for all web services included in our work. Table * shows the relation between the WSDL files and the corresponding components.

| WSDL file | Component Name |
|---|---|
| Myagent.wsdl for agent service | Replica Selection |
| Notification.wsdl for Notification service | Replica Selection |
| Statistics.wsdl for statistics service | Statistics |
| RLSDKS.wsdl for RLS | Node Location Component |
| AliveInfo.wsdl for aliveinfo service | Node Location Component |

## C.1 Myagent.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="Myagent"
targetNamespace="http://www.globus.org/namespaces/myagent/core/Myagent"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://www.globus.org/namespaces/myagent/core/Myagent"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsrp="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-
ResourceProperties-1.2-draft-01.xsd"
    xmlns:wsrpw="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-
ResourceProperties-1.2-draft-01.wsdl"
xmlns:wsdlpp="http://www.globus.org/namespaces/2004/10/WSDLPreprocessor
"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<wsdl:import
    namespace=
    "http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-
ResourceProperties-1.2-draft-01.wsdl"
    location="../../wsrf/properties/WS-ResourceProperties.wsdl" />


<!--============================================================

                     T Y P E S

  ============================================================-->
<types>
<xsd:schema
targetNamespace="http://www.globus.org/namespaces/myagent/core/Myagent"
    xmlns:tns="http://www.globus.org/namespaces/myagent/core/Myagent"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

      <!-- REQUESTS AND RESPONSES -->


        <xsd:element name="fileName" type="xsd:string"/>
        <xsd:element name="filePosition" type="xsd:string"/>
        <xsd:element name="clientPosition" type="xsd:string"/>
```

```
<!-- CLIENT REQUEST -->
<xsd:element name="receiveRequest">
<xsd:complexType>
      <xsd:sequence>
            <xsd:element ref="tns:clientPosition" minOccurs="1"
            maxOccurs="1"/>
            <xsd:element ref="tns:fileName" minOccurs="1"
            maxOccurs="1"/>
            <xsd:element name="roundTripTime" type="xsd:long"
            minOccurs="1" maxOccurs="1"/>
            <xsd:element name="providedSize" type="xsd:long"
            minOccurs="1" maxOccurs="1"/>
      </xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:element name="respondToRequest">
<xsd:complexType>
      <xsd:sequence>
            <xsd:element ref="tns:fileName" minOccurs="1"
            maxOccurs="1"/>
            <xsd:element name="status" type="xsd:string"
            minOccurs="1" maxOccurs="1"/>
            <xsd:element name="roundTripTime" type="xsd:long"
            minOccurs="1" maxOccurs="1"/>
      </xsd:sequence>
</xsd:complexType>
</xsd:element>

<!-- ANT ALGORITHM -->

<xsd:element name="motivate">
<xsd:complexType>
      <xsd:sequence>
            <xsd:element ref="tns:clientPosition" minOccurs="1"
            maxOccurs="1"/>
            <xsd:element name="delegationSite" type="xsd:string"
            minOccurs="1" maxOccurs="1"/>
            <xsd:element name="roundTripTime" type="xsd:long"
            minOccurs="1" maxOccurs="1"/>
            <xsd:element name="rttReq" type="xsd:long"
            minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="tns:fileName" minOccurs="1"
            maxOccurs="1"/>
      </xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:element name="motivateResponse">
      <xsd:complexType/>
</xsd:element>

 <xsd:element name="ant">
       <xsd:complexType>
            <xsd:sequence>
                  <xsd:element name="hops" type="xsd:int"
                  minOccurs="1" maxOccurs="1"/>
```

```
                  <xsd:element name="minRTT" type="xsd:long"
                  minOccurs="1" maxOccurs="1"/>
                  <xsd:element ref ="tns:clientPosition"
                  minOccurs="1" maxOccurs="1"/>
                  <xsd:element name="rttReq" type="xsd:long"
                  minOccurs="1" maxOccurs="1"/>
                  <xsd:element name="minRTTPosition"
                  type="xsd:string" minOccurs="1" maxOccurs="1"/>
                  <xsd:element name="hopStart" type="xsd:string"
                  minOccurs="1" maxOccurs="1"/>
                  <xsd:element name="delegationSite"
                  type="xsd:string" minOccurs="1" maxOccurs="1"/>
                  <xsd:element ref="tns:fileName" minOccurs="1"
            maxOccurs="1"/>
            </xsd:sequence>
      </xsd:complexType>
</xsd:element>
 <xsd:element name="antAck">
      <xsd:complexType/>
</xsd:element>

<xsd:element name="resetMyagentStatus">
      <xsd:complexType/>
</xsd:element>

<xsd:element name="resetMyagentStatusResponse">
      <xsd:complexType/>
</xsd:element>


<!-- FILE INFO -->

<xsd:element name="getFileLists">
      <xsd:complexType/>
</xsd:element>

<xsd:element name="getFileListsResponse" type="xsd:string"/>

<xsd:element name="getFileSize" type="xsd:string"/>
<xsd:element name="getFileSizeResponse" type="xsd:long"/>

<!-- Ping Pong -->

<xsd:element name="getRTT">
      <xsd:complexType>
            <xsd:sequence>
                  <xsd:element name="remotePosition"
                  type="xsd:string" minOccurs="1" maxOccurs="1"/>
                  <xsd:element name="remotePort" type="xsd:int"
                  minOccurs="1" maxOccurs="1"/>
            </xsd:sequence>
      </xsd:complexType>
</xsd:element>

<xsd:element name="getRTTResponse" type="xsd:long"/>
```

```
<!-- Cache -->

<xsd:element name="updateCache">
      <xsd:complexType>
            <xsd:sequence>
                  <xsd:element name="newNodeCache"
                  type="xsd:string" minOccurs="1" maxOccurs="1"/>
                  <xsd:element name="newTimeRecCache"
                  type="xsd:string" minOccurs="1" maxOccurs="1"/>
            </xsd:sequence>
      </xsd:complexType>
</xsd:element>
<xsd:element name="updateCacheResponse">
      <xsd:complexType/>
</xsd:element>

<xsd:element name="getCache">
      <xsd:complexType/>
</xsd:element>
<xsd:element name="getCacheResponse">
      <xsd:complexType>
            <xsd:sequence>
                  <xsd:element name="currentNodeCache"
                  type="xsd:string" minOccurs="0" maxOccurs="1"/>
                  <xsd:element name="currentTimeRecCache"
                  type="xsd:string" minOccurs="0" maxOccurs="1"/>
            </xsd:sequence>
      </xsd:complexType>
</xsd:element>



<!-- RESOURCE PROPERTIES -->

<xsd:element name="StatusClientPosition" type="xsd:string"/>
<xsd:element name="StatusRTTReq" type="xsd:long"/>
<xsd:element name="StatusMinRTT" type="xsd:long"/>
<xsd:element name="StatusFileName" type="xsd:string"/>
<xsd:element name="StatusString" type="xsd:string"/>
<xsd:element name="StatusCounter" type="xsd:int"/>
<xsd:element name="StatusMinRTTPosition" type="xsd:string"/>
<xsd:element name="CacheNodeList" type="xsd:string"/>
<xsd:element name="CacheTimeRec" type="xsd:string"/>

<xsd:element name="MyagentResourceProperties">
<xsd:complexType>
      <xsd:sequence>
            <xsd:element ref="tns:StatusClientPosition"
            minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="tns:StatusRTTReq" minOccurs="1"
            maxOccurs="1"/>
            <xsd:element ref="tns:StatusMinRTT" minOccurs="1"
            maxOccurs="1"/>
            <xsd:element ref="tns:StatusFileName" minOccurs="1"
            maxOccurs="1"/>
```

```
                  <xsd:element ref="tns:StatusString" minOccurs="1"
                  maxOccurs="1"/>
                  <xsd:element ref="tns:StatusCounter" minOccurs="1"
                  maxOccurs="1"/>
                  <xsd:element ref="tns:StatusMinRTTPosition"
                  minOccurs="1" maxOccurs="1"/>
                  <xsd:element ref="tns:CacheNodeList" minOccurs="1"
                  maxOccurs="1"/>
                  <xsd:element ref="tns:CacheTimeRec" minOccurs="1"
                  maxOccurs="1"/>
            </xsd:sequence>
      </xsd:complexType>
      </xsd:element>


</xsd:schema>
</types>


<!--============================================================

                     M E S S A G E S

  ============================================================-->

<message name="MotivateInputMessage">
      <part name="parameters" element="tns:motivate"/>
</message>
<message name="MotivateOutputMessage">
      <part name="parameters" element="tns:motivateResponse"/>
</message>

<message name="Ant">
      <part name="parameters" element="tns:ant"/>
</message>
<message name="AntAckMessage">
      <part name="parameters" element="tns:antAck"/>
</message>

<message name="GetFileListsInputMessage">
      <part name="parameters" element="tns:getFileLists"/>
</message>
<message name="GetFileListsOutputMessage">
      <part name="parameters" element="tns:getFileListsResponse"/>
</message>

<message name="GetFileSizeInputMessage">
      <part name="parameters" element="tns:getFileSize"/>
</message>
<message name="GetFileSizeOutputMessage">
      <part name="parameters" element="tns:getFileSizeResponse"/>
</message>

<message name="GetRTTInputMessage">
      <part name="parameters" element="tns:getRTT"/>
</message>
<message name="GetRTTOutputMessage">
```

```
      <part name="parameters" element="tns:getRTTResponse"/>
</message>

<message name="RequestInputMessage">
      <part name="parameters" element="tns:receiveRequest"/>
</message>
<message name="RequestOutputMessage">
      <part name="parameters" element="tns:respondToRequest"/>
</message>

<message name="ResetMyagentStatusInputMessage">
      <part name="parameters" element="tns:resetMyagentStatus"/>
</message>
<message name="ResetMyagentStatusOutputMessage">
      <part name="parameters"
element="tns:resetMyagentStatusResponse"/>
</message>

<message name="UpdateCacheInputMessage">
      <part name="parameters" element="tns:updateCache"/>
</message>
<message name="UpdateCacheOutputMessage">
      <part name="parameters" element="tns:updateCacheResponse"/>
</message>

<message name="GetCacheInputMessage">
      <part name="parameters" element="tns:getCache"/>
</message>
<message name="GetCacheOutputMessage">
      <part name="parameters" element="tns:getCacheResponse"/>
</message>

<!--============================================================

                     P O R T T Y P E

  ============================================================-->

<portType name="MyagentPortType"
      wsdlpp:extends = "wsrpw:GetResourceProperty"
      wsrp:ResourceProperties="tns:MyagentResourceProperties">

      <operation name="motivate">
            <input message="tns:MotivateInputMessage"/>
            <output message="tns:MotivateOutputMessage"/>
      </operation>

        <operation name="antContainer">
            <input message="tns:Ant"/>
            <output message="tns:AntAckMessage"/>
      </operation>

       <operation name="getFileLists">
            <input message="tns:GetFileListsInputMessage"/>
            <output message="tns:GetFileListsOutputMessage"/>
      </operation>
```

```
       <operation name="getFileSize">
            <input message="tns:GetFileSizeInputMessage"/>
            <output message="tns:GetFileSizeOutputMessage"/>
      </operation>

      <operation name="getRTT">
            <input message="tns:GetRTTInputMessage"/>
            <output message="tns:GetRTTOutputMessage"/>
      </operation>

      <operation name="receiveRequest">
            <input message="tns:RequestInputMessage"/>
            <output message="tns:RequestOutputMessage"/>
      </operation>

      <operation name="resetMyagentStatus">
            <input message="tns:ResetMyagentStatusInputMessage"/>
            <output message="tns:ResetMyagentStatusOutputMessage"/>
      </operation>

      <operation name="updateCache">
            <input message="tns:UpdateCacheInputMessage"/>
            <output message="tns:UpdateCacheOutputMessage"/>
      </operation>

      <operation name="getCache">
            <input message="tns:GetCacheInputMessage"/>
            <output message="tns:GetCacheOutputMessage"/>
      </operation>

</portType>


</definitions>
```

## C.2 Notification.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="Notification"

targetNamespace="http://www.globus.org/namespaces/myagent/core/Notifica
tion"
    xmlns="http://schemas.xmlsoap.org/wsdl/"

xmlns:tns="http://www.globus.org/namespaces/myagent/core/Notification"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsrp="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-
ResourceProperties-1.2-draft-01.xsd"
    xmlns:wsrpw="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-
ResourceProperties-1.2-draft-01.wsdl"
    xmlns:wsntw="http://docs.oasis-open.org/wsn/2004/06/wsn-WS-
BaseNotification-1.2-draft-01.wsdl"

xmlns:wsdlpp="http://www.globus.org/namespaces/2004/10/WSDLPreprocessor
"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```
<wsdl:import
    namespace=
          "http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-
      ResourceProperties-1.2-draft-01.wsdl"
    location="../../wsrf/properties/WS-ResourceProperties.wsdl" />

<wsdl:import
    namespace=
    "http://docs.oasis-open.org/wsn/2004/06/wsn-WS-BaseNotification-
1.2-draft-01.wsdl"
    location="../../wsrf/notification/WS-BaseN.wsdl"/>

<!--============================================================

                    T Y P E S

  ============================================================-->
<types>
<xsd:schema
targetNamespace="http://www.globus.org/namespaces/myagent/core/Notifica
tion"

xmlns:tns="http://www.globus.org/namespaces/myagent/core/Notification"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

      <!-- REQUESTS AND RESPONSES -->
      <xsd:element name="add">
        <xsd:complexType>
            <xsd:sequence>
                  <xsd:element name="cp" type="xsd:string"
                  minOccurs="1" maxOccurs="1"/>
                  <xsd:element name="rttReq" type="xsd:long"
                  minOccurs="1" maxOccurs="1"/>
                  <xsd:element name="minrttp" type="xsd:string"
                  minOccurs="1" maxOccurs="1"/>
                  <xsd:element name="minrtt" type="xsd:long"
                  minOccurs="1" maxOccurs="1"/>
                  <xsd:element name="fileName" type="xsd:string"
            minOccurs="1" maxOccurs="1"/>
            </xsd:sequence>
      </xsd:complexType>
      </xsd:element>

        <xsd:element name="addResponse">
            <xsd:complexType/>
      </xsd:element>

      <!-- RESOURCE PROPERTIES -->

      <xsd:element name="Information" type="xsd:string"/>


      <xsd:element name="NotificationResourceProperties">
      <xsd:complexType>
            <xsd:sequence>
                  <xsd:element ref="tns:Information" minOccurs="1"
                  maxOccurs="1"/>
```

```
            </xsd:sequence>
      </xsd:complexType>
      </xsd:element>

</xsd:schema>
</types>


<!--============================================================

                     M E S S A G E S

  ============================================================-->

<message name="AddInputMessage">
      <part name="parameters" element="tns:add"/>
</message>
<message name="AddOutputMessage">
      <part name="parameters" element="tns:addResponse"/>
</message>


<!--============================================================

                     P O R T T Y P E

  ============================================================-->
<portType name="NotificationPortType"
    wsdlpp:extends="wsrpw:GetResourceProperty
wsntw:NotificationProducer"
    wsrp:ResourceProperties="tns:NotificationResourceProperties">

      <operation name="add">
            <input message="tns:AddInputMessage"/>
            <output message="tns:AddOutputMessage"/>
      </operation>

</portType>


</definitions>
```

## C.3 Statistics. wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="Statistics"

targetNamespace="http://www.globus.org/namespaces/myagent/core/Statisti
cs"
    xmlns="http://schemas.xmlsoap.org/wsdl/"

xmlns:tns="http://www.globus.org/namespaces/myagent/core/Statistics"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsrp="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-
ResourceProperties-1.2-draft-01.xsd"
    xmlns:wsrpw="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-
ResourceProperties-1.2-draft-01.wsdl"
```

```
xmlns:wsdlpp="http://www.globus.org/namespaces/2004/10/WSDLPreprocessor
"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<wsdl:import
    namespace=
    "http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-
ResourceProperties-1.2-draft-01.wsdl"
    location="../../wsrf/properties/WS-ResourceProperties.wsdl" />


<!--============================================================

                     T Y P E S

  ============================================================-->
<types>
<xsd:schema
targetNamespace="http://www.globus.org/namespaces/myagent/core/Statisti
cs"

xmlns:tns="http://www.globus.org/namespaces/myagent/core/Statistics"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

      <!-- REQUESTS AND RESPONSES -->
        <xsd:element name="fileName" type="xsd:string"/>
        <xsd:element name="filePosition" type="xsd:string"/>
      <xsd:element name="rttReq" type="xsd:long"/>
      <xsd:element name="clientPosition" type="xsd:string"/>

      <xsd:element name="collectAccessInfo">
        <xsd:complexType>
            <xsd:sequence>
                  <xsd:element ref="tns:fileName" minOccurs="1"
                  maxOccurs="1"/>
                  <xsd:element ref="tns:filePosition" minOccurs="1"
                  maxOccurs="1"/>
            </xsd:sequence>
      </xsd:complexType>
      </xsd:element>

        <xsd:element name="collectAccessInfoResponse">
            <xsd:complexType/>
      </xsd:element>

      <xsd:element name="addClient">
       <xsd:complexType>
            <xsd:sequence>
                  <xsd:element ref="tns:clientPosition" minOccurs="1"
            maxOccurs="1"/>
                  <xsd:element ref="tns:rttReq" minOccurs="1"
            maxOccurs="1"/>
            </xsd:sequence>
      </xsd:complexType>
      </xsd:element>
```

```
      <xsd:element name="addClientResponse">
            <xsd:complexType/>
      </xsd:element>

      <xsd:element name="removeClient" type="xsd:string"/>
      <xsd:element name="removeClientResponse">
            <xsd:complexType/>
      </xsd:element>

      <xsd:element name="reset">
            <xsd:complexType/>
      </xsd:element>
      <xsd:element name="resetResponse">
            <xsd:complexType/>
      </xsd:element>


</xsd:schema>
</types>


<!--============================================================

                     M E S S A G E S

  ============================================================-->

<message name="CollectAccessInfoInputMessage">
      <part name="parameters" element="tns:collectAccessInfo"/>
</message>
<message name="CollectAccessInfoOutputMessage">
      <part name="parameters" element="tns:collectAccessInfoResponse"/>
</message>

<message name="AddClientInputMessage">
      <part name="parameters" element="tns:addClient"/>
</message>
<message name="AddClientOutputMessage">
      <part name="parameters" element="tns:addClientResponse"/>
</message>

<message name="RemoveClientInputMessage">
      <part name="parameters" element="tns:removeClient"/>
</message>
<message name="RemoveClientOutputMessage">
      <part name="parameters" element="tns:removeClientResponse"/>
</message>

<message name="ResetInputMessage">
      <part name="parameters" element="tns:reset"/>
</message>
<message name="ResetOutputMessage">
      <part name="parameters" element="tns:resetResponse"/>
</message>


<!--============================================================
```

```
                        P O R T T Y P E

  ============================================================-->

<portType name="StatisticsPortType">

      <operation name="collectAccessInfo">
            <input message="tns:CollectAccessInfoInputMessage"/>
            <output message="tns:CollectAccessInfoOutputMessage"/>
      </operation>

      <operation name="addClient">
            <input message="tns:AddClientInputMessage"/>
            <output message="tns:AddClientOutputMessage"/>
      </operation>

      <operation name="removeClient">
            <input message="tns:RemoveClientInputMessage"/>
            <output message="tns:RemoveClientOutputMessage"/>
      </operation>

      <operation name="reset">
            <input message="tns:ResetInputMessage"/>
            <output message="tns:ResetOutputMessage"/>
      </operation>

</portType>


</definitions>
```

## C.4 RLSDKS.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="RLSDKS"
targetNamespace="http://www.globus.org/namespaces/myagent/core/RLSDKS"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://www.globus.org/namespaces/myagent/core/RLSDKS"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsrp="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-
ResourceProperties-1.2-draft-01.xsd"
    xmlns:wsrpw="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-
ResourceProperties-1.2-draft-01.wsdl"

xmlns:wsdlpp="http://www.globus.org/namespaces/2004/10/WSDLPreprocessor
"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<wsdl:import
    namespace=
    "http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-
ResourceProperties-1.2-draft-01.wsdl"
    location="../../wsrf/properties/WS-ResourceProperties.wsdl" />


<!--============================================================
```

```
                    T Y P E S

  ============================================================-->
<types>
<xsd:schema
targetNamespace="http://www.globus.org/namespaces/myagent/core/RLSDKS"
    xmlns:tns="http://www.globus.org/namespaces/myagent/core/RLSDKS"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">


      <!-- REQUESTS AND RESPONSES -->

        <xsd:element name="fileName" type="xsd:string"/>
      <xsd:element name="filePosition" type="xsd:string"/>

      <xsd:element name="add">
        <xsd:complexType>
            <xsd:sequence>
                  <xsd:element ref="tns:fileName" minOccurs="1"
                  maxOccurs="1"/>
                  <xsd:element ref="tns:filePosition" minOccurs="1"
                  maxOccurs="1"/>
            </xsd:sequence>
      </xsd:complexType>
      </xsd:element>
        <xsd:element name="addResponse">
            <xsd:complexType/>
      </xsd:element>

        <xsd:element name="delete">
        <xsd:complexType>
            <xsd:sequence>
                  <xsd:element ref="tns:fileName" minOccurs="1"
                  maxOccurs="1"/>
                  <xsd:element ref="tns:filePosition" minOccurs="1"
                  maxOccurs="1"/>
            </xsd:sequence>
      </xsd:complexType>
      </xsd:element>
        <xsd:element name="deleteResponse">
            <xsd:complexType/>
      </xsd:element>


      <xsd:element name="query" type="xsd:string"/>
      <xsd:element name="queryResponse">
      <xsd:complexType>
            <xsd:sequence>
                  <xsd:element ref="tns:fileName" minOccurs="1"
                  maxOccurs="1"/>
                  <xsd:element ref="tns:filePosition" minOccurs="0"
                  maxOccurs="unbounded"/>
            </xsd:sequence>
      </xsd:complexType>
      </xsd:element>
```

```
      <xsd:element name="getFirstDKSNodeURL">
            <xsd:complexType/>
      </xsd:element>

      <xsd:element name="getFirstDKSNodeURLResponse"
      type="xsd:string"/>

      <xsd:element name="getSuccessor">
            <xsd:complexType/>
      </xsd:element>

      <xsd:element name="getSuccessorResponse" type="xsd:string"/>

      <xsd:element name="getRoutingTable">
            <xsd:complexType/>
      </xsd:element>

      <xsd:element name="getRoutingTableResponse">
      <xsd:complexType>
            <xsd:sequence>
                  <xsd:element name="myIP" type="xsd:string"
                  minOccurs="1" maxOccurs="1"/>
                  <xsd:element name="routingTable" type="xsd:string"
                  minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
      </xsd:complexType>
      </xsd:element>

</xsd:schema>
</types>


<!--============================================================

                     M E S S A G E S

  ============================================================-->
<message name="AddInputMessage">
      <part name="parameters" element="tns:add"/>
</message>
<message name="AddOutputMessage">
      <part name="parameters" element="tns:addResponse"/>
</message>

<message name="DeleteInputMessage">
      <part name="parameters" element="tns:delete"/>
</message>
<message name="DeleteOutputMessage">
      <part name="parameters" element="tns:deleteResponse"/>
</message>

<message name="QueryInputMessage">
      <part name="parameters" element="tns:query"/>
</message>
<message name="QueryOutputMessage">
      <part name="parameters" element="tns:queryResponse"/>
</message>
```

```
<message name="GetFirstDKSNodeURLInputMessage">
      <part name="parameters" element="tns:getFirstDKSNodeURL"/>
</message>
<message name="GetFirstDKSNodeURLOutputMessage">
      <part name="parameters"
element="tns:getFirstDKSNodeURLResponse"/>
</message>

<message name="GetSuccessorInputMessage">
      <part name="parameters" element="tns:getSuccessor"/>
</message>

<message name="GetSuccessorOutputMessage">
      <part name="parameters" element="tns:getSuccessorResponse"/>
</message>

<message name="GetRoutingTableInputMessage">
      <part name="parameters" element="tns:getRoutingTable"/>
</message>

<message name="GetRoutingTableOutputMessage">
      <part name="parameters" element="tns:getRoutingTableResponse"/>
</message>


<!--============================================================

                      P O R T T Y P E

  ============================================================-->

<portType name="RLSDKSPortType">

      <operation name="add">
            <input message="tns:AddInputMessage"/>
            <output message="tns:AddOutputMessage"/>
      </operation>

      <operation name="delete">
            <input message="tns:DeleteInputMessage"/>
            <output message="tns:DeleteOutputMessage"/>
      </operation>

      <operation name="query">
            <input message="tns:QueryInputMessage"/>
            <output message="tns:QueryOutputMessage"/>
      </operation>

       <operation name="getFirstDKSNodeURL">
            <input message="tns:GetFirstDKSNodeURLInputMessage"/>
            <output message="tns:GetFirstDKSNodeURLOutputMessage"/>
      </operation>

       <operation name="getSuccessor">
            <input message="tns:GetSuccessorInputMessage"/>
            <output message="tns:GetSuccessorOutputMessage"/>
```

```
      </operation>

       <operation name="getRoutingTable">
            <input message="tns:GetRoutingTableInputMessage"/>
            <output message="tns:GetRoutingTableOutputMessage"/>
      </operation>

</portType>


</definitions>
```

## C.5 AliveInfo.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="AliveInfoService"

targetNamespace="http://www.globus.org/namespaces/myagent/core/AliveInf
oService"
    xmlns="http://schemas.xmlsoap.org/wsdl/"

xmlns:tns="http://www.globus.org/namespaces/myagent/core/AliveInfoServi
ce"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsrp="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-
ResourceProperties-1.2-draft-01.xsd"
    xmlns:wsrpw="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-
ResourceProperties-1.2-draft-01.wsdl"

xmlns:wsdlpp="http://www.globus.org/namespaces/2004/10/WSDLPreprocessor
"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<wsdl:import
    namespace=
    "http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-
ResourceProperties-1.2-draft-01.wsdl"
    location="../../wsrf/properties/WS-ResourceProperties.wsdl" />


<!--============================================================

                     T Y P E S

  ============================================================-->
<types>
<xsd:schema
targetNamespace="http://www.globus.org/namespaces/myagent/core/AliveInf
oService"

xmlns:tns="http://www.globus.org/namespaces/myagent/core/AliveInfoServi
ce"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

      <!-- RESOURCE PROPERTIES -->

      <xsd:element name="ServicesAddress" type="xsd:string"/>
```

```
      <xsd:element name="AliveInfoResourceProperties">
      <xsd:complexType>
            <xsd:sequence>
                  <xsd:element ref="tns:ServicesAddress" minOccurs="1"
                  maxOccurs="1"/>
            </xsd:sequence>
      </xsd:complexType>
      </xsd:element>

</xsd:schema>
</types>


<!--================================================================

                        P O R T T Y P E

  ================================================================-->
<portType name="AliveInfoPortType"
    wsdlpp:extends="wsrpw:GetResourceProperty
                wsrpw:GetMultipleResourceProperties"
    wsrp:ResourceProperties="tns:AliveInfoResourceProperties">


</portType>
</definitions>
```

# Appendix D User Guides

There are four directories under thesis work package, named (1) `DataConsistency`, including the source codes for data consistency component; (2)`DongLi_Thesis_JavaCode`, including the source code for services and clients; (3) `Gar`, including the gar package for deploying services; and (4) `schema` including the .wsdl file.

To set up the first Grid node, you have to re-build the RLS service instead of using the `Gar` file in the `Gar` directory. Firstly, copy the `RLSDKS.java` in `org/globus/myagent/services/core/tmp_RLSDKS/1/` to the RLSDKS `impl` directory and then build it. The corresponding command is:

```
./globus-build-service.sh –d
org/globus/myagent/services/core/RLSDKS/ -s
schema/RLSDKS/RLSDKS.wsdl
```

Secondly, deploy aliveinfo service. Third, run the command,

```
$GLOBUS_LOCATION/bin/mds-servicegroup-add -s
https://ServiceAddress/wsrf/services/DefaultIndexServiceEntr
y -z none alive-position-registration.xml
```

To deploy a service in the GT4 web container, like agent service, run the command

```
$GLOBUS_LOCATION/bin/globus_deploy_gar
Gar/org_globus_myagent_services_core_agent.gar
```

The clients for services access are under the directory `DongLi_ThesisWork/DongLi_Thesis_JavaCode/org/globus/myagent/clients/agent/`. To add an item in the RLS and query a file, run the command `java org.globus.myagent.clients.agent.RLSDKSClient RLSDKS ***`, where "***" should be replaced with the service address.

To submit replica selection request, run

```
java org.globus.myagent.clients.agent.GenClientRequest.
```

To listen to the selection result, run

```
java -DGLOBUS_LOCATION=$GLOBUS_LOCATION
org.globus.myagent.clients.agent.GenClientListener n
```

where n should be replace by the relica selection requests number.

To run the data consistency component, first run the command `fam` as the superuser and then run the `run-example-d.sh ***`, where *** should be replaced by the directory name you would like to monitor.