Chapter 3

Semantic Meta Data (SMD): An Approach to Next Generation Knowledge Centric Web / Grid Services

As Web Services have matured, they have been substantially leveraged within the academic, research and business communities. The Grid is an emerging platform to support on demand “virtual organizations” for coordinated resource sharing and problem solving on a global scale. Web/Grid services, metadata and semantics are becoming increasingly important for service sharing and effective reuse.

In this chapter we present a novel approach to next generation knowledge centric Web/Grid Services, with the ultimate purpose of facilitating interoperability, automation and knowledgeable reuse of services for problem solving. We used the Globus Toolkit, which is an open source software toolkit used for building Grid systems and applications.

3.1 Introduction

Our modern and mobile society depends on fast and stable access to information and computing resources, at all time and in all places. Grid systems enable this seamless access and allow us to achieve better qualitative and quantitative results by coordinating the resources we depend on. Additionally, grid systems are designed to
give a more secure and more dependable access to these resources. Grids are defined as persistent environments that enable software applications to integrate. In practice a grid is an infrastructure to control sharing of resources, i.e., being able to access others resources and giving access to own resources to others. In the vision of the designers of grid computing the resources necessary for an application are accessed using standardized access methods through a grid infrastructure. The metaphor “grid” comes from power grids, where electrical current can be accessed in a seamless way using a standardized plug. The www can be seen as a specialized predecessor of a grid, which allows access to documents. In general the accessible services include resources like computing cycles, special computing capabilities, storage, devices or even human expertise.

The Grid is “flexible, secure, coordinated resource sharing among dynamic collections of individuals, institutions and resources - what we refer to as virtual organizations.” [42].

Large scale science and engineering are undertaken through the interaction of people, heterogeneous computing resources, information systems and instruments, all of which are geographically and organizationally dispersed. “The Grid” is an emerging platform to support coordinated resource sharing and problem solving on a global scale for data intensive and compute intensive applications [42]. The Grid was originally focused on sharing computational power and resources for advanced science and engineering. The ‘metacomputing’ projects of the early 1990s set out to build virtual supercomputers
using networked computer systems. The target applications were primarily continue to be, large scale science. For increasing the computational power by combining large numbers of geographically diverse systems raises the issues of scalability and heterogeneity. Scalability brings a number of challenges, the inevitability of failure of components, the need for automation, the need to exploit the locality of resources due to network latency and the increasing number of organizational boundaries, emphasizing authentication and trust issues. Larger scale applications may also result from the composition of other applications, which increases the complexity of systems. Heterogeneity is addressed by middleware, such as the Globus Toolkit [44], to provide uniformity through a standard set of interfaces to the underlying resources.

Early Grid middleware exploits a range of protocols such as LDAP for directory services and file store queries [45], GridFTP for large scale reliable data transfer and SSL for security. Higher level functionality, such as tolerant scalable data replication [46], exploit these. Some attention has been paid to data intensive rather than compute intensive Grid use. However, research and development activities relating to the Grid have generally focused on applications where data was stored in files and there is little support for transactions, relational database access or distributed query processing [47].

The Grid community is now actively developing fundamental mechanisms for the interaction of any kind of resource including
documents, databases, instruments, archives and people. Support for data interaction is focused on consistent access to databases from Grid applications [48].

3.2 Metadata

The rapid increase in the number and variety of resources on the World Wide Web has made inappropriateness of traditional schemas of resource description for web resources and has encouraged significant activities recently on defining web compatible schemas named "metadata".

Metadata, in general, is defined as 'data about data' or 'information about information'. In the other words, metadata is data that describe information resources. Metadata is data that provide extra information about other data. For example, a photo can be described using the following metadata: <dateTaken> 01/01/2011 </dateTaken>, <placeTaken> seminar room </placeTaken> and <whatAbout> project meeting </whatAbout>.

The information being described by metadata, may be considered at the first look as corporal and digital information resources such as books, newspapers, journals, photographs and so on. Greenberg [3] refers to this data as "object" and states that this object is any entity, form or mode for which contextual data can be recorded. The universe of objects to which metadata can be applied is radically diverse and
seemingly endless, ranging from corporeal and digital information resources, such as a monograph, newspaper or photograph, to activities, events, persons, places structures, transactions, relationships, execution directions and programmatic applications.

Metadata, therefore, captures the wide range of intrinsic or extrinsic information about a variety of objects. These intrinsic or extrinsic characteristics and features are described in the individually structured data elements that facilitate object use, identification and discovery. Semantic metadata refers to the metadata that are formally modeled based on their context, thus giving them meaning.

Metadata exist at all levels of the Grid, ranging from low level repositories of resource handles to upper level application related services. At the time of writing, the metadata of low level hardware related Grid services is stored and managed by core Grid services such as Globus MDS and RGMA.

The way that current service oriented infrastructure handles and manages services metadata is not adequate and effective for metadata to help services discovery and knowledge sharing. There are no problems for humans to understand XML based metadata because we know the meaning of these English words, the question is: “can machines understand and consume them?” so that they can perform automatic processing with regards to the use of Web/Grid services. Clearly without further assumptions, the answer will be no. The
Semantic Web/Grid are extensions of the current Web/Grid in which information and services are given well defined meaning, better enabling computers and people to work in cooperation. We believe that the first step towards the Semantic Web/Grid is to make the Web/Grid full of rich SMD, in other words, metadata with semantics.

3.3 Ontology Design

Being the conceptual models that capture domain knowledge, ontologies can be looked upon for aiding meaningful information retrieval. Ontology is a common understanding of a domain, which is often perceived as a collection of concepts, relationships and instances.

Classes are the focus of most ontologies. Classes describe concepts in the domain. For example, a class of wines represents all wines. Specific wines are instances of this class. A class can have subclasses that represent concepts that are more specific than the superclass. For example, we can divide the class of all wines into red, white and rosé wines. Alternatively, we can divide a class of all wines into sparkling and non sparkling wines.

In practical terms, developing ontology includes:

- Defining classes in the ontology.
- Arranging the classes in a taxonomic (subclass–superclass) hierarchy.
- Defining slots and describing allowed values for these slots.
- Filling in the values for slots for instances.
Separating the domain knowledge from the operational knowledge is another common use of ontologies. We can describe a task of configuring a product from its components according to a required specification and implement a program that does this configuration independent of the products and components themselves. We can then develop ontology of PC components and characteristics and apply the algorithm to configure made to order PCs. We can also use the same algorithm to configure elevators if we “feed” elevator component ontology to it.

3.3.1 Concept Map Generation from Owl Ontologies

Concept map generation is very much needed for knowledge assessment tools, which are time and effort consuming. A lot of ontologies written in OWL are available on Internet and could be reused as a base for concept maps. Therefore these resources may be effectively used if corresponding algorithms and tools are available. Both ontologies and concept maps represent some domain. Both have classes or concepts and relations between them. Unlike concept maps ontologies have also attributes for classes, their values and restrictions on them. In other words ontologies are more expressive. At the same time concept maps also could represent the same, more expressive features using only concepts and links between them. Since ontologies are described using special languages, such as OWL, DAML+OIL, Ontolingua or special knowledge structure like frames,
correspondence between ontology elements and concept map elements should be defined.

Mappings between main elements of OWL ontology and elements of the concept map are shown in Figure 3.1. Main elements of OWL ontology which correspond to the concept are a class, an instance, a data type property, its value and a type of value, while an object property corresponds to a link. An object property is semantic relation between classes or instances.

Fig 3.1: Correspondence of main ontology elements to concept map elements
Table 3.1: Correspondence of specific OWL elements to concept map elements

<table>
<thead>
<tr>
<th>#</th>
<th>OWL elements</th>
<th>Concept map Elements</th>
<th>#</th>
<th>OWL elements</th>
<th>Concept map elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>owl:Class</td>
<td>Concept</td>
<td>7</td>
<td>owl:FunctionalProperty</td>
<td>Link</td>
</tr>
<tr>
<td>2</td>
<td>owl:DatatypeProperty</td>
<td>Concept</td>
<td>8</td>
<td>owl:InverseFunctionalProperty</td>
<td>Link</td>
</tr>
<tr>
<td>3</td>
<td>owl:DeprecatedClass</td>
<td>Concept</td>
<td>9</td>
<td>owl:ObjectProperty</td>
<td>Link</td>
</tr>
<tr>
<td>4</td>
<td>owl:DeprecatedProperty</td>
<td>Link</td>
<td>10</td>
<td>owl:SymmetricProperty</td>
<td>Link</td>
</tr>
<tr>
<td>5</td>
<td>owl:equivalentClass</td>
<td>Class</td>
<td>11</td>
<td>owl:TransitiveProperty</td>
<td>Link</td>
</tr>
<tr>
<td>6</td>
<td>owl:equivalentProperty</td>
<td>Link</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Along with links retrieved from object properties additional links are introduced:

- “is a” to represent hierarchal relation between two classes.
- “is instance of” to represent relation between a class and its instances.
- “is synonym” to represent that two classes/instances are synonyms.
- “is not” to represent that two classes are complement.
- “has property” to present relation between a class/instance and a datatype property.
- “has type” to present relation between a datatype property and a data type of its value.
- “has value” to present relation between a datatype property and its value.

Here a concept map generation algorithm is specified. Information mined from ontology is stored in an incidence matrix, where names of concepts and relations between concepts (the names of the links and direction of the links) are stored. An example of an abstract concept map is shown in Figure 3.2 a. The concrete example for the abstract
concept map is given in Figure 3.2b. The corresponding incidence matrix for the abstract concept map is given in Table 3.2.

![Abstract Concept Map](image)

Fig: a)

![Concrete Concept Map](image)

Fig: b)

Fig 3.2: Example of concept maps a) abstract, b) concrete

<table>
<thead>
<tr>
<th>Name of concept</th>
<th>Concept 1</th>
<th>Concept 2</th>
<th>Concept 3</th>
<th>Concept 4</th>
<th>Instance</th>
<th>Property</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concept 1</td>
<td>link</td>
<td></td>
<td>Is synonym</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concept 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>has property</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concept 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concept 4</td>
<td>Is synonym</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instance</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>has instance of</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Property</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>has type</td>
<td>has value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Basic steps for concept map generation from ontology are the following:

1. Read an ontology file and check OWL syntax.

2. Find all classes (begin creating of incidence matrix).

3. Find subclasses of each class (for particular class add link “is a” which goes from subclass to super class in the matrix).

4. For each class check equivalence and similarity (add link “is synonym” in the matrix).

5. For each class check intersection and union to other classes (add link “is a” in the matrix).

6. For each class check complement relations to other classes (add link “is not” in the matrix).

7. Find instances of each class (add instances and links “is instance of” between appropriate classes and instances which go from a instance to a class to the matrix).

8. Find datatype properties for each class and instance (add properties and links “has property” between appropriate class/instance and a property to the matrix).

9. Find values for each datatype property (add properties values and links between a datatype property and its value “has value” to the matrix).

10. Find types for datatype property’s values (add a value’s type and a link between a datatype property’s value and its type to the matrix).

11. Find object properties for each class/instance (add appropriate links between classes or instances to the matrix).

12. Check if an object property is symmetric or transitive (extend the matrix with appropriate links).
13. Perform correction of concept and link names (replace under strike sign “_” with space).

14. Display completed incidence matrix as a graph.

At first, we choose ontology for concept map generation and then the concept map generation software performs all transformations and displays a generated concept map. Before transformation actions verification of chosen *.owl file is performed to check correctness of used OWL syntax. If the ontology doesn’t confirm OWL syntax, then an error message is received. If verification is successful the concept map generation algorithm starts. In the beginning all classes, their hierarchy and instances are found, afterwards data type properties, their values and types are found and at the end object properties are found. Finally, correction of concept and link names is performed and generated concept map is displayed.

For simpler implementation previously mentioned algorithm’s steps are merged in the following way: transformation of ontology classes and instances (steps 2, 3, 4, 5, 6, 7), transformation of datatype properties, their values and types (steps 8, 9, 10), transformation of object properties (steps 11, 12). The first step of the concept map generation algorithm is to find a class hierarchy and its extension with instances of classes (Figure 3.3).
At first the root class is found and a class name is added to the incidence matrix. Then it is checked if the class has instances. If yes, then names of instances are added to the matrix as well as the link “is instance of” to relate instances with the root class. If the root class hasn’t instances, it is checked if it has subclasses. If yes, then the
names of subclasses are added to the matrix as well as the link “is a” to relate the root class with subclasses. Each subclass is checked if it has instances and subclasses similarly as in case with the root class. Based on the results of checking the matrix is updated. Instances of the class are mined before subclasses due to simpler recursion programming needed for mining. After all classes have been found they are checked to mine additional information. Classes are checked if they have synonym classes. If yes, then the matrix is updated with the link “is synonym” between synonym classes. Then classes are checked if they are result of intersection or union of other classes, if yes, then in case of intersection the link “is a” is added from the class to other classes from intersection, in case of union the link “is a” is added from classes from union to the class.

The next step is to find datatype properties, their values and types of values (Figure 3.4). First, datatype properties are found for each class. The matrix is updated with names of datatype properties, as well as the link “has property” at the appropriate class. Then for each datatype property it is checked if allowed values for this property are defined. If there are such values, their names are added to the matrix, as well as the link “has value” at the datatype property. If the type of values is defined, it is also added to the matrix and the link “has type” at the appropriate datatype property. For each class which has datatype property it is checked, some of defined allowed values assigned to its instances. In case if any of values are assigned to the instance the link between the class and this value is deleted and
added between the instance and the value, i.e., the value becomes local, it is related only to the instance.

---

Fig 3.4: Transformation of datatype properties, their values and types of values
Checking of object property’s symmetry and transitivity provides the concept map with additional links, that is, in case of the symmetric link, the link in the concept map is bidirectional and in case of transitive link the third inferred link is added to the concept map. Second, object properties between instances are found. Then the matrix is updated with links between instances. However, links between superclasses of instances are deleted. Characteristic of symmetry and transitivity is checked, too. The realization of this step is shown in Figure 3.5.

Fig 3.5: Transformation of object properties
3.4 Ontology Centric Approach to SMD Management

Ontology is defined as an explicit specification of a conceptualization. If a program wants to compare conceptual information across two knowledge bases on the Web, it has to know when any two given terms are being used to mean the same thing. It intends to capture all metadata of Web/Grid services and the concepts related to domain in which these services operate. It further models these metadata, concepts and their relations in a structure using commonly agreed terms. The purpose is to abstract the ontological entities of metadata and put them in context, thus giving them meaning. Our framework uses ontologies to perform metadata and context modeling in which entities such as services will be conceptualized as ontological concepts and an entity’s metadata will be conceptualized as its properties. Context modeling will conceptualize all other entities related to the concerned entity and establish relations among them via concepts properties. Overall context modeling will create a self contained ontology in which metadata can be interpreted unambiguously by both humans and machines. Ontology based metadata and context modeling provides a common communication language for Web/Grid service providers and consumers.

The key features of the approach are as follows: Firstly, ontologies are used for metadata and context modeling, thus help towards interoperability and machine understandability. Secondly, knowledge
acquisition, i.e., service metadata collection and semantics tagging, is carried out semi automatically through a formal knowledge binding process also known as semantic annotation. Thirdly, Web ontology languages are used for SMD knowledge representation, thus enabling knowledge sharing and effective reuse.

### 3.4.1 Deployment details

**Thin Client:** In this configuration (Fig. 3.6), the ontological knowledge is made available via Web based protocols. Such a system would be appropriate for use with applications that require to reference standard ontology knowledge.

![Diagram of Thin Client](image)

**Fat Client:** A fat client configuration provides local persistence for ontological knowledge that the application generates (Fig. 3.7). Depending on the configuration details, knowledge may be stored locally as files or in a database for increased performance and
reliability. The local persistent storage can also provide local caching for ontological knowledge accessed via Web.

**Ontology Server API**

<table>
<thead>
<tr>
<th>Ontology Directory</th>
<th>Inference Engine</th>
<th>Query Optimizer</th>
<th>Working Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Connector</td>
<td>Web Connector</td>
<td></td>
<td>Internet</td>
</tr>
</tbody>
</table>

**Fig 3.7: Fat Client**

*Ontology Client/Server Architecture:* This configuration models a full fledged ontology management system with a complex set of functionalities. To provide ontology sharing and evolution among a large number of clients, the ontology management functions would be moved to an ontology server (Fig. 3.8). Not only does this arrangement provide for easier ontologies sharing and updating in a distributed environment, but it also allows for additional optimization and optimized indexing to support better knowledge query and retrieval. Within the knowledge Grid scenario, the Ontology Server (OS) provides the basic semantic interoperability functionalities. In fact, it provides the knowledge producer with the possibility of interacting with heterogeneous and distributed document repositories. It guarantees to the knowledge providers the necessary autonomy to organize the
managed contents space. From the conceptual point of view, the OS is the important server since it manages the schema for the stored knowledge expressed using OWL/RDF and determines the interactions with the other semantic web or Grid servers and components.

![Ontology Server API](image)

Fig 3.8: Ontology Client-Server Architecture

### 3.4.2 Inference Engines

Inference Engine provides reasoning capabilities for service SMD management system. It has two main usages: first an inference engine can be used to help construct a large ontology by performing actions
such as subsumption, classification, concept consistency check and second, to discover a specific service in terms of user query criteria an inference engine is needed to reason against the SMD repositories. There are different ontological reasoning engines. For instance, the FaCT reasoner can perform terminological reasoning, the RACER reasoner can perform instance reasoning.

**3.5 Web services with Grid Environment**

Web services are self contained, self describing software applications that can be published, discovered, located and invoked across the web, using standard web protocols. Nowadays, the basic support for web services is provided through the SOAP (Simple Object Access Protocol), WSDL (Web Service Description language) and UDDI (Universal Description, Discovery and Integration) specifications, which address message format, service description, service publishing and lookup. Foster and Kesselman proposed distributed computing infrastructure for advanced science and engineering, called as Grid. This word was initially coined from the electricity power grid. In recent years the focus has shifted away from the high performance aspect towards definition of the grid problem a flexible, secure, coordinated resource sharing among dynamic collection of individuals, institutions and resources, whatever we refer to as virtual organizations. The main focus of Grid was to obtain high performance.

Once ontology is being created and queried it can be enhanced to “Grid Environment”, where more than one domain can be merged to
have a single ontology, which will be able to query any information about merged domain as shown in fig 3.9 and proposed methodology to grid environment shown in fig 3.10. Web Services and Grid Computing are technologies that go hand in hand these days. Grid is a very dynamic environment because new services may be added into it or may cease to exist.

Web Services

(Greater Computation)

Next Generation Web

(Rich Semantics with valid information)

Grid Services

Fig 3.9: Two Services which lead to Next Generation Web

Fig 3.10: Proposed Methodology to Grid Environment
At times, the queries may require combining Ontologies which are very complex. Then creation of the grid environment may take weeks. The idea of creating a grid was to share the computations and resources across the web. Grid infrastructure started to leverage existing Web service standards with some extensions in order to build Grid based systems on top of a pure Service Oriented Architecture (SOA). Today, we can share anything across the web using grid like images, databases, computations, people, archives etc. The grid components require information about the functionality and interfaces and this information must have an agreed meaning and this will be supported by metadata. This has been the key for Grid services.

3.6 Setting up a Grid Infrastructure

In this section we discuss the various Tools, Standards and Platforms required for the setting up a grid infrastructure.

3.6.1 Beowulf

Beowulf Clusters are scalable performance clusters based on commodity hardware, on a private network, with open source software (Linux) infrastructure. Beowulf is a multi computer architecture which can be used for parallel computations. It is a system which usually consists of one server node and one or more client nodes connected together via Ethernet or some other network. It is a system built using
commodity hardware components, like any PC capable of running a Unix/Linux like operating system, with standard Ethernet adapters and switches. Beowulf also uses commodity software like the Linux or Solaris operating system, Parallel Virtual Machine (PVM) and Message Passing Interface (MPI). The server node controls the whole cluster and serves provides file service to the client nodes. It is also the cluster's console and gateway to the outside world. Large Beowulf machines might have more than one server node and possibly other nodes dedicated to particular tasks, for example consoles or monitoring stations. In most cases client nodes in a Beowulf system are dumb. Nodes are configured and controlled by the server node and do only what they are told to do. In a disk less client configuration, client nodes don’t even know their IP address or name until the server tells them what it is.

One of the main differences between Beowulf and a Cluster of Workstations (COW) is the fact that Beowulf behaves more like a single machine rather than many workstations. In most cases client nodes do not have keyboards or monitors and are accessed only via remote login or possibly serial terminal. Beowulf nodes can be thought of as a CPU + memory package which can be plugged in to the cluster, just like a CPU or memory module can be plugged into a motherboard. Beowulf is a technology of clustering computers to form a parallel, virtual supercomputer. Although there are many software packages such as kernel modifications, PVM and MPI libraries and configuration tools which make the Beowulf architecture faster, easier
to configure and much more usable, one can build a Beowulf class
machine using standard Linux distribution without any additional
software[52].

### 3.6.2 Condor

Condor is a resource management system designed to support high
throughput computations by discovering idle resources on a network
and allocating those resources to application tasks [51]. Condor is
technology that combines existing resources to provide a considerable
amount of computational power. This may be of benefit if application
is parallelized. As a result, you may save your time and effort. The
effort required to parallelize your application is little since you do not
need to rewrite your code in a different language. You can run many
different applications in Condor environment that were written in
different languages such as Java, C or Matlab. Condor also has a
checkpoint system which is able to resume your process from the last
saved state [51].

High throughput computing aims to provide a fault tolerant
execution environment for long running processes whilst utilizing
efficiently the resources available on the Internet. Condor, a high
throughput computing system, offers the following features:

- Class Ads: A framework to match the resources with the
  specified job descriptions.
- Job Checkpoint and Migration: For some particular
  applications, it is possible to resume the application from its
last state using a checkpoint file. This provides fault tolerance. For example, in the case of a failure in a machine, the job can be safely transferred to another machine.

- Remote System Calls: Condor supports I/O related jobs (processes, executables) which require processing input files and generating output files. By using this way, the files will automatically be transferred to the remote machines, hence you are not required to transfer the files manually by yourself or have a shared file system.

3.6.3 Parallel and Distributed Application (MPICH2)

For running an application in Parallel and Distributed environment we need a supporting environment which can run our programs. These libraries are Message Passing Interface (MPI) libraries, which allow many computers to communicate with one another. It is used in computer clusters and supercomputers. MPI is a language independent communications protocol used to program parallel computers. Both point to point and collective communications are supported. MPI is a message passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation. MPI's goals are high performance, scalability and portability. MPI remains the dominant model used in high performance computing today. Most MPI implementations consist of a specific set of routines (i.e., an API) callable from Fortran, C or C++ and from any language capable of
interfacing with such routine libraries. The advantages of MPI over older message passing libraries are portability (because MPI has been implemented for almost every distributed memory architecture) and speed (because each implementation is in principle optimized for the hardware on which it runs) [53].

Various tools are available in market which provides the implementation of MPI libraries. Some of them are OpenMP, Open MPI, MPICH, etc. MPICH is a freely available, portable implementation of MPI, a standard for message passing [54].

### 3.6.4 Resource Manager (TORQUE)

Efficient use of cluster is possible if we use a Resource Manager for managing the jobs, also known as Job Manager or Job Scheduler. They must have some basic features like Interfaces which helps to define workflows and/or job dependencies, Automatic submission of executions, Interfaces to monitor the executions, Priorities and/or queues to control the execution order of unrelated jobs. The main concept of any Resource Manager is to manage Jobs and Dependencies and Users request. Beyond the basic, single OS instance scheduling tools there are two major architectures that exist for Job Scheduling software.

1. **Master/Agent architecture**: The historic architecture for Job scheduling software. The Job Scheduling software is installed on a single machine (Master) while on production machines only a small
component (Agent) is installed that awaits commands from the Master, executes them and returns the exit code back to the Master.

2. **Cooperative architecture**: A decentralized model where each machine is capable of helping with scheduling and can offload locally scheduled jobs to other cooperating machines. This enables dynamic workload balancing to maximize hardware resource utilization and high availability to ensure service delivery [55].

Scheduling jobs and allocating resources on a cluster quickly becomes a challenge once more than a few users start running codes on the system. Manual coordination of runs is tedious, particularly when different codes have very different resource requirements. A job queuing and scheduling facility solves these problems by automatically executing jobs as resources become available ensuring optimal utilization of the cluster. Moreover, a good job scheduler can be configured to enforce operational policies about when and where jobs belonging to different users may be run [56]. Various Resource Managers are available such as OpenPBS, Sun Grid Engine (SGE), Torque, Maui Cluster Scheduler, TITAN etc.

While TORQUE has a built-in scheduler, pbs_sched, it is typically used solely as a resource manager with a scheduler making requests to it. Resources managers provide the low level functionality to start, hold, cancel and monitor jobs. Without these capabilities, a scheduler alone cannot control jobs. While TORQUE is flexible enough to handle scheduling, it is primarily used in batch systems. Pooling resources in a batch system typically reduces technical administration
of resources while offering a uniform view to users. Once configured properly, batch systems abstract away many of the details involved with running and managing jobs, allowing higher resource utilization. For example, users typically only need to specify the minimal constraints of a job and do not need to know the individual machine names of each host on which they are running. With this uniform abstracted view, batch systems can execute thousands and thousands of jobs simultaneously. Batch systems are comprised of four different components: (1) Master Node (2) Submit/Interactive Nodes (3) Compute Nodes and (4) Resources.

1. **Master Node:** A batch system will have a master node where pbs_server runs. Depending on the needs of the systems, a master node may be dedicated to this task or it may fulfill the roles of other components as well.

2. **Submit/Interactive Nodes:** Submit or interactive nodes provide an entry point to the system for users to manage their workload. For these nodes, users are able to submit and track their jobs. Additionally, some sites have one or more nodes reserved for interactive use, such as testing and troubleshooting environment problems. These nodes have client commands (such as qsub and qhold).

3. **Compute Nodes:** Compute nodes are the workhorses of the system. Their role is to execute submitted jobs. On each compute node, pbs_mom runs to start, kill and manage submitted jobs. It communicates with pbs_server on the master node. Depending on the
needs of the systems, a compute node may double as the master node (or more).

4. **Resources**: Some systems are organized for the express purpose of managing a collection of resources beyond compute nodes. Resources can include high speed networks, storage systems, license managers and so forth. Availability of these resources is limited and needs to be managed intelligently to promote fairness and increased utilization.

The life cycle of a job can be divided into four stages: (1) creation (2) submission (3) execution and (4) finalization.

1. **Creation**: Typically, a submit script is written to hold all of the parameters of a job. These parameters could include how long a job should run (walltime), what resources are necessary to run and what to execute.

2. **Submission**: A job is submitted with the qsub command. Once submitted, the policies set by the administration and technical staff of the site dictates the priority of the job and therefore, when it will start executing.

3. **Execution**: Jobs often spend most of their lifecycle executing. While a job is running, its status can be queried with qstat.

4. **Finalization**: When a job completes, by default, the stdout and stderr files are copied to the directory where the job was submitted [57].
3.6.5 WS GRAM

GT4 (Globus Toolkit 4) contains a web service based Grid Resource Allocation and Management (GRAM) component. WS GRAM is a WSRF based web service used to remotely submit, monitor and cancel jobs. The jobs can be managed by schedulers like Condor, PBS etc.

![Diagram of WS GRAM Protocol Steps]

Fig 3.11: Major Protocol steps in WS GRAM

3.6.5.1 Managed Job Factory Service

The Managed Job Factory Service (MJFS) is used to create an instance of the stateful Managed Job Service by calling the createManagedJob method. The resulting Managed Job Service is used to control and monitor a job. In addition the MJFS publishes information about the characteristics of the compute resource using the WS Resource specification. The following are the operations provided by Managed job factory.
- createManagedJob: This operation creates a Managed Job Service instance, subscribes the client for notifications if requested and replies with one or two endpoint references (EPRs). These EPRs in turn point at the actual Managed Job Service used to control and monitor a job.

- getResourceProperty, getMultipleResourceProperties, and QueryResourceProperties are all part of the WS ResourceProperties portType.

### 3.6.5.2 Managed Job Service

Managed Job Service provides release, SetTerminationTime, destroy, subscribe, getResourceProperty, getMultipleResourceProperty operations.

### 3.6.5.3 Job Description

A GRAM job can be described using the XML Schema.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<job>
  <executable>/bin/echo</executable>
  <directory>${GLOBUS_USER_HOME}</directory>
  <argument>Welcome to the CDAC PSE.</argument>
  <stdout>${GLOBUS_USER_HOME}/stdout</stdout>
  <stderr>${GLOBUS_USER_HOME}/stderr</stderr>
</job>
```

or this is more complex example which demonstrates how to submit a multijob:

```xml
<?xml version="1.0" encoding="UTF-8"?>
  xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing">
  <factoryEndpoint>
    <wsa:Address>
      https://localhost:8443/wsrf/services/ManagedJobFactoryService
    </wsa:Address>
  </factoryEndpoint>
</multiJob>
```
<wsa:Address>https://localhost:8443/wrsf/services/ManagedJobFactoryService</wsa:Address>
<wsa:ReferenceProperties>
  <gram:ResourceID>Fork</gram:ResourceID>
</wsa:ReferenceProperties>
<factoryEndpoint>
<executable>/bin/date</executable>
<stdout>${GLOBUS_USER_HOME}/stdout.p1</stdout>
<stderr>${GLOBUS_USER_HOME}/stderr.p1</stderr>
<count>1</count>
</job>

<job>
<factoryEndpoint>
<executable>/bin/echo</executable>
<argument>Hello World!</argument>
<stdout>${GLOBUS_USER_HOME}/stdout.p2</stdout>
<stderr>${GLOBUS_USER_HOME}/stderr.p2</stderr>
<count>1</count>
</job>
</multiJob>

Fig 3.12: WS GRAM Job Description

3.7 Implementation

1. Beowulf Cluster of 4 nodes is formed and /home/mpich folder is shared on all 3 client nodes using Network File System (NFS). Along with it password less Secure Shell (ssh) is allowed in this cluster.
BEOWULF CLUSTER

Fig 3.13: Beowulf Cluster of 4 nodes

2. MPICH2 is installed on this shared file system.

MPI Libraries

Fig 3.14: Beowulf Cluster of 4 nodes with MPI Libraries
3. After installing MPICH2, Torque is installed for managing jobs in Beowulf Cluster. A TORQUE cluster consists of one head node and many compute nodes. The head node runs the pbs_server and pbs_sched daemon and the compute nodes run the pbs_mom daemon. Users submit jobs to pbs_server using the qsub command. When pbs_server receives a new job, it informs the scheduler. When the scheduler finds nodes for the job, it sends instructions to run the job with the node list to pbs_server. Then, pbs_server sends the new job to the first node in the node list and instructs it to launch the job. This node is designated the execution host and is called Mother Superior. Other nodes in a job are called sister moms.

![TORQUE Diagram](image)

Fig 3.15: Beowulf Cluster of 4 nodes with TORQUE

4. After installing a Job scheduler (TORQUE), Globus is installed and TORQUE is then configured with GRAM for job submission through Globus. A Certificate Authority (CA) is also configured, since in order to work within Globus and GSI enabled tools, all
users and services need to have a certificate issued from a trusted certificate authority (CA).

This whole Infrastructure forms one grid site in next step we’ll see how to configure this site with Grid Infrastructure.

![GRID Infrastructure with GRAM](image)

**Fig 3.16: GRID Infrastructure with GRAM**

5. Finally our local site GRAM is then configured with CONDOR a meta scheduler. Condor and Globus are complementary technologies, as demonstrated by CondorG, a Globus enabled version of Condor that uses Globus to handle inter-organizational problems like security, resource management for
supercomputers and executable staging. Condor can be used to submit jobs to systems managed by Globus and Globus tools can be used to submit jobs to systems managed by Condor. The Condor and Globus teams work closely with each other to ensure that the Globus Toolkit and Condor software fit well together.

End user submits job through Internet using a grid portal, while submitting the job he also request the resources required by the job. Meta Scheduler then queries the Monitoring and Discovery System (MDS). The Monitoring and Discovery System is a suite of web services to monitor and discover resources and services on Grids. This system allows users to discover what resources are considered part of a Virtual Organization (VO) and to monitor those resources. MDS then gives him the index of sites and resources available in those sites. Then he interacts with the local Job Manager of the site and finally the requested job is executed.
Fig 3.17: Grid Infrastructure with GRAM, Meta Scheduler
3.8 Conclusion

Creating and populating rich semantic metadata on the Web/Grid have been commonly accepted as the route leading to the Semantic Web/Grid vision. This chapter describes our effort towards the next generation service oriented computing infrastructure with rich metadata and semantic support. It presents an integrated framework for SMD management for Web/Grid services.

Publications:
