1-1-2006

An agent-based peer-to-peer grid computing architecture

Jia Tang
University of Wollongong, jt989@uow.edu.au

Minjie Zhang
University of Wollongong, minjie@uow.edu.au

An agent-based peer-to-peer grid computing architecture

Abstract

The conventional computing Grid has developed a service oriented computing architecture with a superlocal resource management and scheduling strategy. This architecture is limited in modeling computer systems with highly dynamic and autonomous computing resources due to its server-based computing model. The super-local resource management and scheduling strategy also limits the utilization of the computing resources. In this paper, we propose a multi-agent based Peer-to-Peer Grid computing architecture. This novel architecture solves the above issues, while provides reasonable compatibility and interoperability with the conventional Grid systems and clients. The main characteristics of this architecture are highlighted by its promising performance and scalability, and its adaptive resource management and scheduling mechanisms. With this architecture, it is promising to build large scale high performance commodity computing Grids at low cost.

Keywords

agent, based, peer, peer, grid, computing, architecture

Disciplines

Physical Sciences and Mathematics

Publication Details

Zhang, M. & Tang, J. 2006, "An agent-based peer-to-peer grid computing architecture", Australasian Symposium on Grid Computing and e-Research (now AusPDC), Australian Computer Society, Sydney, Australia, pp. 33-39.

An Agent-based Peer-to-Peer Grid Computing Architecture

Convergence of Grid and Peer-to-Peer Computing

Jia Tang **Minjie Zhang**

School of Information Technology and Computer Science
University of Wollongong,
North Fields Ave, Wollongong, NSW 2500,
Email: {jt989, minjie}@uow.edu.au

Abstract

The conventional computing Grid has developed a service oriented computing architecture with a super-local resource management and scheduling strategy. This architecture is limited in modeling computer systems with highly dynamic and autonomous computing resources due to its server-based computing model. The super-local resource management and scheduling strategy also limits the utilization of the computing resources. In this paper, we propose a multi-agent based Peer-to-Peer Grid computing architecture. This novel architecture solves the above issues, while provides reasonable compatibility and interoperability with the conventional Grid systems and clients. The main characteristics of this architecture are highlighted by its promising performance and scalability, and its adaptive resource management and scheduling mechanisms. With this architecture, it is promising to build large scale high performance commodity computing Grids at low cost.

Keywords: Grid Computing, Peer-to-Peer Computing, Resource Management, Scheduling, Multi Agent Systems.

1 Introduction

In the past few decades, the architectures for computing systems have made significant advances. The *client/server* (C/S) architecture (Goldman, Rawles & Mariga 1999) was the first success as an alternative to the conventional mainframe systems. It shifts the processing burden to the client computer, and therefore improves the overall efficiency (Alfred 2003). Later, we saw the rise of LAN-based *cluster computing* (Pfister 1997) in 1980s and WAN-based *metacomputing* (Smarr & Catlett 1992) in 1990s, both of which derive from the client/server architecture, and aim at sharing workload further through computer networks. Inspired by the power grid, *Grid computing* (Foster & Kesselman 1999) exploits cluster computing and metacomputing further to Internet-scale computing resource sharing, selection, and aggregation.

While computing Grids have been widely used in computational science, *Peer-to-Peer computing* (P2P) (Barkai 2002) has achieved wide prominence in the context of multimedia file exchange. It uses the computing power at the edge of a connection rather than within the network. The client/server architecture does not exist in a peer-to-peer system. Instead, peer nodes act as both clients and servers - their roles are determined by the characteristics of the tasks and the status of the system. This architecture minimizes the workload per node, and maximizes the utilization of the overall computing resources among the network.

Today, the shear numbers of desktop systems make the potential advantages of interoperability between desktops and servers into a single Grid system quite compelling. However, these commodity systems have significantly different properties than the conventional server-based Grid systems. They are usually highly autonomous and heterogeneous systems. And their availability varies from time to time.

The conventional computing Grid has developed a service-oriented computing architecture with a super-local resource management and scheduling strategy. In Globus Tookit 4 (Globus Alliance 2005) (GT4, the official implementation of the current Grid standards), nine high-level Grid services defined by Open Grid Service Architecture (OGSA) (Foster, Kesselman, Nick & Tuecke 2002) are implemented using Web Services mechanisms to provide functionalities such as resource management, scheduling, etc. As these services are required to be stateful and Web Services are usually stateless, Web Services Resource Framework (WSRF) (Foster, Czajkowski, Ferguson, Frey, Graham, Maguire, Snelling & Tuecke 2005) was introduced so that the stateful information can be preserved as WS-Resources between different service invocations.

In GT4, Monitoring and Discovery System (MDS) (Fitzgerald, Foster, Kesselman, Laszewski, Smith & Tuecke 1997) and Grid Resource Allocation and Management (GRAM) (Czajkowski, Foster, Karonis, Kesselman, Martin, Smith & Tuecke. 1998) work as the resource management and the job management services respectively. MDS manages the monitoring and discovering of the resources. It gets the information from several information providers and publishes it to other services. Three of the information providers are related to job execution: two for gathering date related to cluster resources, and one for providing the information about the local schedulers. GRAM manages the submission and execution of the jobs. It uses a super-local scheduling strategy: the super scheduler schedules a job to a suitable local scheduler based on the job's requirements and the local schedulers' statuses provided by MDS; the local scheduler then schedules the job to a specific computing node. Figure 1 (Foster 2005) depicts this strategy. The dashed area indicates the service host (i.e. the super scheduler). The compute element consists of a local scheduler and computing nodes.

As Web Services provide standard means for communications and object invocations between the clients and the service providers, the embrace of Web Services increases the interoperability of the Grid. The super-local scheduling strategy was also a success

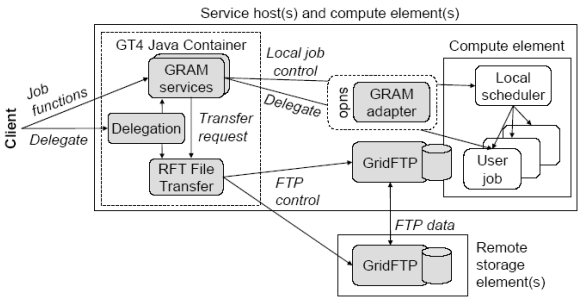

Figure 1: The Super-local Scheduling Strategy in GT4

in high-end computational environments, because of its flexibility in the face of various widely accepted local schedulers such as Condor (Epema, Livny, van-Dantzig, Evers & Pruyne 1996). But in order to implement and deploy a Grid made up of commodity systems, the autonomous, heterogeneous, and highly dynamic nature of such an environment must be carefully considered. These properties further lead to the following issues of the conventional Grid:

1. WSRF was developed as a complement to Web Services in order to make stateless Web Services stateful. However, it can result in significant overhead on the network traffic and the object invocations due to the transmissions of the WS-Resources between the client and the service host, and the conversions between the internal states of a service and their WS-Resource equivalents.

2. The current service oriented architecture has poor adaptability in terms of performance, availability, and scalability, as no facility has been provided by the current Grid systems to allow automatic deployment of the services according to the clients' requests and the load of the Grid.

3. The dependence on the local schedulers increases the complexity of application programming in the Grid environment, as it is difficult to provide various local schedulers with a uniform programming interface that supports task decomposition, state persistence, and inter-task communications.

4. The super-local resource management and scheduling strategy intensively relies on the underlying local schedulers. This "two commit" process leads to more complex handling on resource discovering, selection, and allocation compared with a one-level process. The lack of direct management of the computing nodes can cause unsuitable selection of resources, and unbalanced loads, and therefore limits the overall performance. In addition, as new computing nodes can only join the local clusters instead of join the Grid directly, the scalability of the local schedulers highly affects the overall scalability of the Grid.

5. It is not feasible to introduce local schedulers into our targeted environment, as local schedulers require a relatively static and non-autonomous environment.

This paper attempts to tackle the above issues by proposing a new task model and applying peer-to-peer computing model to the Grid architecture. The rest of this paper is structured as follows. Section 2 proposes a multi-agent (Lesser 1999) based peer-to-peer Grid computing architecture, demonstrates its new task model, and explains how to exploit the peer-to-peer computing model to build a commodity computing GridS. Section 3 probes into the compatibility and interoperability issues with the existing Grid systems and clients. Section 4 concludes this paper.

2 An Agent-based Peer-to-Peer Grid Architecture

In this section, we propose S.M.A.R.T-2 (Service-oriented, Microkernel, Agent-based, Rational, and Transparent), a multi-agent (Lesser 1999) based P2P Grid computing architecture with an adaptive resource management and scheduling strategy. We demonstrate its overall architecture and core components first. Then we discuss its job/service model. In the end, we probe into its peer-to-peer computing architecture as well as its resource management and scheduling mechanisms.

2.1 Overall Architecture and Core Components

There are two kinds of entities in S.M.A.R.T-2: the clients and the computing nodes (or called *peers*). A *client* is defined as a generic computing device that seeks services from the Grid using Web Services standards. A *computing node* is the place where tasks are executed and computing occurs. Each computing node runs a microkernel Grid container, which serves as the runtime and managerial environment for jobs and services. A computing device can serve as a client and a peer at the same time.

A task (i.e. a job or a service) is described as a group of linked modules in S.M.A.R.T-2. A *module* is the fundamental unit that can be scheduled among the peers. All modules run on the peers, more specifically, within the S.M.A.R.T-2 Grid containers. Figure 2 shows the relationship between the modules and the container.

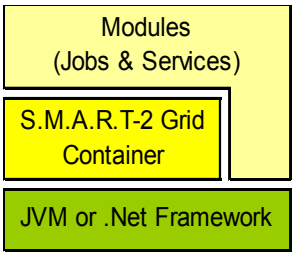

Figure 2: Components within a S.M.A.R.T-2 Computing Node

The S.M.A.R.T-2 Grid container allows the modules to register to the service portal as Web Services. The *service portal* conforms to Web Services standards (W3C 2002), and allows the clients to interact with the Grid using SOAP messages. Figure 3 demonstrates the overall architecture of the S.M.A.R.T-2 Grid container.

Inside the container, there are four components: the runtime environment (RT), the management agent (MA), the profiling agent (PA), and the computing agent (CA). The runtime environment provides the fundamental routines and the runtime libraries for both the agents and the modules. For example, the XML parsing libraries, and the implementations of Web Services standards such as SOAP and WSDL are included in the runtime environment; the service portal is also part of the runtime environment. The management agent provides the managerial interfaces between the container and the Grid Management Service. It manages the container, the

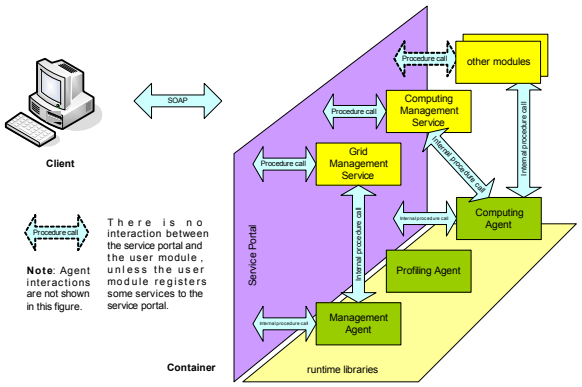

Figure 3: Schematic View of S.M.A.R.T-2

policies and the configurations as well. The profiling agent gathers the status of the network, the peers, and the running modules, and provides optimized dynamic configurations for the computing agent. The computing agent is responsible for managing the lifecycle of the modules, locating the resources and the modules, discovering the services, and scheduling the modules and the service innovations among the peers while providing fault-tolerance and load balance. Figure 4 depicts the agent interactions within the Grid container.

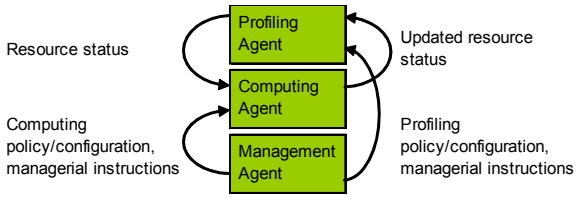

Figure 4: Agent Interactions in S.M.A.R.T-2 Container

Besides these components, there are two predefined modules which register as *Grid Management Service* (GMS) and *Computing Management Service* (CMS) respectively. GMS allows the users who have certain privileges to manage the Grid, e.g. specifying the computing policy/configuration, and monitoring the status of the Grid. CMS provides the interfaces for the clients to manage the computing resources. In S.M.A.R.T-2, all objects involved in the computing process are regarded as resources. These resources include the executables of the modules, the service descriptions that the modules register, the data files, the storage, the CPU, etc.

2.2 Job/Service Model

As mentioned in previous section, S.M.A.R.T-2 uses modules to describe jobs and services. A module consists of the module description, the executables, the serialization, and the module owned files. Figure 5 shows the composition of a module.

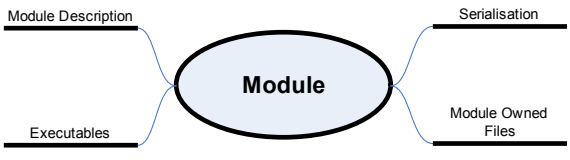

Figure 5: S.M.A.R.T-2 Module

The Module Description (MD) has two sections: the task section and the service section.

Figure 6 depicts the task section of MD. This section defines the task related information, and consists of two subsections which are listed as follows:

1. The deployment description subsection defines the information of a module's executables (e.g. what is the entry point of the module if it is a startup module), and the dependencies of that module. A module's dependency is another module or a service that the module depends on.

2. The computing policy subsection defines a module's (a) minimum hardware requirements on a peer's machine type, processor type, and contributed cycle/memory/storage amount; (b) estimated amount of computation; (c) expected completion time; (d) priority level; and (e) relay policies).

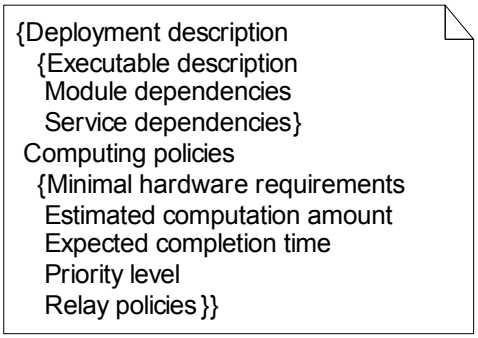

Figure 6: Task Section of the Module Description

The service section of MD is optional and is only needed if the module registers one or more services to the Grid. It uses WSDL and WSRF to define the service interfaces and related stateful information respectively.

The executables are Java bytecode files or .NET executables. When a running module is suspended by a user or if it is relocated, it is serialized. This process is equivalent to the class serialization (Sun Microsystems Inc. 2004) of Java. It allows the Grid container to store the runtime dynamics of the module, and restore them when the module is resumed. The module owned files (MOFs) are the files that tightly bind to a module. These files are regarded as part of the module, and migrate along with the module's description, executables and serialization.

A group of linked modules consists of a complete task. Each module implements a fraction of the overall task. As these modules can be executed at the same time on different peers, load balance and parallelism are achieved. Each task has a startup module. After all modules of a task have been deployed to the Grid, the client can start the task through CMS. CMS then uses the *create* method of the IModuleContext interface to create an instance of the startup module. Once the startup module is instantized and runs, it can start instances of other modules using the same interface. Figure 7 depicts the hierarchy of the module instances in S.M.A.R.T-2

In S.M.A.R.T-2, whenever a module is instantized, it gets access to the IModuleContext interface, which is provided by the computing agent. This interface defines three kinds of methods which respectively allow a module's instance to (a) create instances of other modules, (b) perform procedure calls (i.e. invoke methods of other modules), and (c) delete used module instances to release their occupied resources.

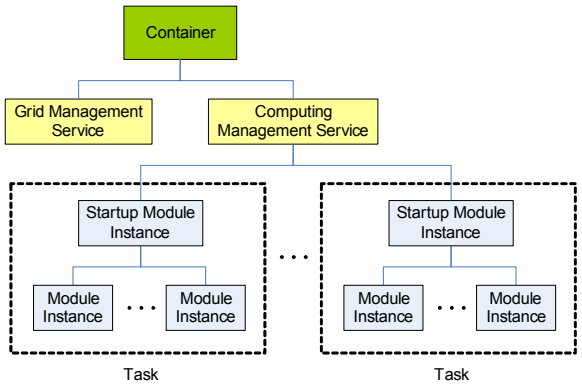

Figure 7: Hierarchy of the Module Instances in S.M.A.R.T-2

As a task must be able to expand to multiple peers, S.M.A.R.T-2 allows a module instance to be created remotely (i.e. on another peer). Table 1 shows the definition of the IModuleContext interface.

```
public interface IModuleContext {

  public ModuleInstance
    create(String moduleName,
           Object... args)
    throws ModuleException;

  public Object
    invoke(ModuleInstance moduleInstance,
           Object... args)
    throws ModuleException;

  public void
    delete(ModuleInstance moduleInstance)
    throws ModuleException;

  /**
   * For static method only
   */
  public Object
    invoke(String moduleName,
           String method,
           Object... args)
    throws ModuleException;
}
```

Table 1: IModuleContext Interface

2.3 Peer-to-Peer Computing Architecture

The interconnected peers comprise the S.M.A.R.T-2 Grid. The peers which have a relatively large number of connections are called *hubs*. When the Grid is constructed, a number of computing nodes which have high availability, good connectivity, and good performance are selected as the backbone of the Grid. Each of them connects to at least two of the others permanently. As new nodes appear, they register to at lease one of the backbone nodes so that a bidirectional connection can be made between the new nodes and their registered backbone nodes.

In S.M.A.R.T-2, a connection is directional (i.e. "A connects to B" does not presume "B connects to A"). In addition, it is not equivalent to a network connection. If A can successfully originate a network connection to B, then A has a connection to B. The connections of a peer are represented by a hash table, where the endpoints of the connections are the keys, and the objects representing the strength of the connections (called simulated synapses) are the values. The following values are defined for a simulated synapse.

1. `strength`, whose value is (0, 1], represents the current strength of the connection. A value "1" means that the connection is a permanent connection.

2. `deathThreshold`, whose value is randomly selected from a user configured range, when a connection is created. When `strength` is less than `deathThreshold`, the connection is removed from the hash table, i.e. the connection breaks up.

3. `activateThreshold`. When a connection is created, a random value is selected from a user configured range as `activateThreshold`, and a random initial value is given to `strength`. At that stage, the connection is inactive. Afterwards, when `strength` grows to a value greater than `activateThreshold`, the connection becomes active, and the `activateThreshold` is set to 0.

4. `permThreshold`. If an active connection's `strength` continues growing to a value greater than `permThreshold`, `strength` is set to 1, and the connection becomes a permanent connection.

Two operations can be applied to a synapse: the *grow* operation, which increase the strength of the connection; and the *decay* operation, which decrease the strength of the connection. Table 2 shows these operations.

An `impulse` defines four fields: `type_ttl`, `serial`, `from` and `message`. Assume that O represents the peer which generates the message, and R represents any of the peers which reply to O. An impulse transmitted from O to R is called an outbound impulse. An impulse transmits from R to O is called an inbound impulse. For any outbound message, the value of `type_ttl` indicates the Time-To-Live (TTL) of the impulse, and is set by O when O creates the impulse; the `serial` field contains a unique number generated by O; the from field is set to O; and the `message` field contains the actual message carried by the impulse. When R replies to O, it resets `type_ttl` to -1 to indicate that the impulse carries a replied message; `serial` is not changed; from is reset to R; and `message` is set to the replied message.

When a peer starts, a fixed size queue which is used to cache the impulses relayed by the peer is created. `Hashtable<Long, List<Impulse>>` impulses is also created to store the inbound impulses, where the key (whose value is `Long`) denotes the `serial` of the impulse, and the value (which is a list of `Impulse`) denotes the inbound impulses. When a relay process starts, an outbound impulse is created by O with its fields being set, and an empty list is created and put into the hash table. Then O transmits the impulse to all of its active connections. When any of the peers receives the impulse, it checks whether the impulse is already in its queue. If true, it discards the impulse; otherwise it decreases the TTL by one, and then checks whether TTL equals 0. If it does, the impulse is discarded; otherwise the peer appends the impulse to the end of the queue, and relays the impulse to all of its active connections. Finally it checks whether it is able to respond to the message carried by the impulse. If true, an outbound impulse will be generated and transmitted directly to O. Table 3 demonstrates the relay process.

```
private static
  Hashtable<Peer, Synapse> synapses;
public static void grow(Peer peer) {
  Synapse synapse = synapses.get(peer);
  if(synapse == null)
    synapses.put(peer,
      Synapse.createTempSynapse());
  else {
    if(synapse.strength == 1)
      return;
    if(synapse.activateThreshold == 0) {
      synapse.strength =
        Math.pow(synapse.strength, 0.5);
      if(synapse.strength >
         synapse.permThreshold)
        synapse.strength = 1;
    }
    else {
      synapse.strength +=
       Math.pow(synapse.activateThreshold,
                2);
      if(synapse.strength >
         synapse.activateThreshold) {
        synapse.strength =
          synapse.deathThreshold +
          (synapse.permThreshold -
           synapse.deathThreshold) *
          GOLDEN_SECTION;
        synapse.activateThreshold = 0;
      }
    }
  }
}
public static void decay(Peer peer) {
  Synapse synapse = synapses.get(peer);
  if(synapse.strength == 1)
    return;
  if(synapse.activateThreshold == 0)
    synapse.strength =
      Math.pow(synapse.strength, 2);
  else
    synapse.strength -=
      Math.pow(synapse.activateThreshold,
               2);
  if(synapse.strength <
     synapse.deathThreshold)
    synapses.remove(peer);
}
```

Table 2: Operations on Simulated Synapse

After O transmits the impulse, it suspends the calling thread for a period of time specified before the transmission or until the number of replies reaches a threshold. Whenever a reply comes back from R to O, and there exists a corresponding list in the hash table, it is added to the list, and the grow operation is performed on the connection to R. When the thread is resumed, the replies are retrieved from the corresponding list in the hash table. Then O goes through all its connections, and performs the decay operation on the connections without a reply. Afterwards, all replies are returned to the thread for selection. Table 4 shows the `getReplies` method, which is implemented in CA.

With the selection process (see Section 2.4), the relay process enables load balance and the election of the most suitable peer for a certain payload (i.e. the message). In the long run, the connections between the peers are optimized according to the characteristics of the payload. News hubs are also developed so

```
impulse = CA.receiveImpulse();

if(impulse.isReply()) {
  List<Impulse> list =
    impulses.get(impulse.getSerial());
  if(list != null) {
    list.add(impulse);
    grow(impulse.getFrom());
  }
}
else if(queue.indexOf(impulse) == -1) {
  Handler handler =
    Container.getHandler(
      impulse.getMessage());
  if(--type_ttl > 0) {
    appends impulse to the queue
    CA.relay(impulse);
  }
  if(handler != null) {
    Message reply =
      handler.handle(
        impulse.getMessage());
    impulse.type_ttl = -1;
    Peer dest = impulse.from;
    impulse.from = Container.getLocal();
    impulse.message = reply;
    CA.send(impulse, dest);
  }
}
/**
 * else
 *   Discard impulse
 */

impulse.destroy();
```

Table 3: Relay Process

that the Grid will gain better connectivity and higher ratio of resource utilization, and work more efficiently.

2.4 Resource Management and Scheduling Mechanisms

S.M.A.R.T-2 uses *resource matrices* to track the status of the computing resources. Figure 5 depicts a sample of the matrix. It defines the type of the resource, where the resource resides, and the resource's status (called profile) or the description of the resource. A peer's local resources are registered by the profiling agent when the peer starts. The profiling agent also updates the profiles of the local resources when they change. Figure 8 depicts a sample definition of the processor profile.

When a module requires a resource, its container C may match the required resource with those in the resource matrix first. If none of them matches the requirement, the container starts a relay process. Or the container may start the replay process immediately when receiving the module's request. How the container behaves is determined by the type of the resource. For example, the local service resources have precedence over the remote service resources, but there is no such difference in terms of processor resources.

During the relay process, the participated peers look up the required resource in their resource matrices. If matching resources exist, the references to these resources are returned to C. If multiple replies exist, C starts a resource selection process to determine which resource is the most suitable one. The

```
public static List<Impulse>
  getReplies(Impulse impulse) {
  List<Impulse> list =
    impulses.remove(impulse.getSerial());
  ArrayList<Peer> peers =
    new ArrayList<Peer>(synapses.size());
  peers.addAll(synapses.keySet());
  for(Impulse i : list)
    peers.remove(i.getFrom());
  for(Peer peer : peers)
    decay(peer);
  return list;
}
```

Table 4: getReplies Method

Resource Type	Residing Peer	Resource Profile/Description	References
Processor	192.168.2.1	Pentium-4, 1.8G, fully contributed, no running module	N/A
Storage	192.168.2.1	1024M Free Space, Transfer Speed 160Mbps/120Mbps (Read/Write)	N/A
Module Instance	192.168.2.1	name = decrypt, id = 1234	192.168.2.7 192.168.2.8
File	192.168.2.2	/modules/decrypt.mar	192.168.2.1
Service	192.168.2.4	/unix-encrypt, Module Description with Service Section	192.168.2.1

Table 5: A Sample Resource Matrix of Peer 192.168.2.1

outcome is then returned to the module for its subsequent operations. And if the type of the located resource has local precedence, it will be cached in the resource matrix of C. A resource matrix only caches a limited number of references. Each time a cached reference is retrieved, it is regarded as "updated". The least updated entry will be removed if the cache is full and a new reference comes in.

```
{Capability
  {Machine type
   Processor type
   Contributed cycles amount
   Contributed memory amount
   Contributed storage amount}
 Load
  {Module#1
   Module#2
   ...}}
```

Figure 8: A Sample Processor Profile Definition

Once the reference to a resource is obtained, the resource is accessible to the module through S.M.A.R.T-2. Each time a resource is accessed its reference is quoted and passed to the resource's residing peer R. Then R will perform the actual operations and send the results back to the module. When the module finishes using the resource, it notifies R so that the resource can be released on R.

A peer also keeps records of other peers which have cached references to its local file, service, and module instance resources, so that the references can be updated when the actual resources migrate to other peers.

In S.M.A.R.T-2, files can be uploaded to the backbone nodes through CMS. Unlike other resources, local files never appear in the resource matrix. When a file is located and used, it can be cached by the peer that uses the file, if there is sufficient storage on that peer.

The executables of any modules is regarded as files, and need to be uploaded to the Grid before the execution of the module. S.M.A.R.T-2 has a two-stage scheduling mechanism. Once a module is uploaded, its residing peer O will trigger a relay process informing other peers the potential workload. Other peers will rely to O if they can execute the module. O then determines the suitability of these peers (including O). The module will be moved to the winner if the winner is a backbone node; otherwise it is transferred to the winner and cached there. At the second stage, when the module is about to be created. A relay process will be started to locate the module. Once it is located, it will be scheduled and executed by its residing peer.

3 Discussion

In this section, we discuss the compatibility and interoperability issues with the existing Grid systems and clients.

Recalling the job/service model, it is easy to find that the new model enables the modeling of both the conversional stateless services and stateful tasks. A module is allowed to register its own services to the service portal using Web Services standards. Hence, any WS-compatible client is capable of accessing these services through S.M.A.R.T-2. There are two means to maintain stateful information for a service in S.M.A.R.T-2. The client and the service can use agreed methods, e.g. WS-Resource, to exchange the stateful information. S.M.A.R.T-2 supports WS-Resource standards, hence a WS-Resource based client needs no modification to work with S.M.A.R.T-2 as long as the service interface is not changed. Another way to preserve the states throughout different service innovation transactions is to create transaction-specific service module. In this case, a token representing a certain transaction is passed in the service innovations. When a new transaction starts, the startup module of the service creates a new service module to serve the transaction. The stateful information is maintained by the service modules. The tokens act as the identifiers for the startup module to dispatch the service innovations to an appropriate service module. Once the transaction is done, the client implicitly notifies the service's startup module so that the startup module can delete the corresponding service module and release the resources.

As S.M.A.R.T-2 conforms to Web Services and WS-Resource standards, any module in is able to operate on the services provided by other WS-compatible Grids using these standards. However, being different in the architecture and the programming model, S.M.A.R.T-2 has neither the binary compatibility nor the source code compatibility for the programs running in the existing Grids.

4 Conclusion

In this paper, we firstly analyzed the service-oriented architecture and the server-based computing model

of the conventional Grid, and designed a new computing architecture. By applying a new job/service model and a peer-to-peer computing model, the new architecture is more efficient and flexible when dealing with open systems. Secondly, we introduced an adaptive resource management and scheduling framework to achieve load balance and high ratio of resource utilization. Finally, we clarified how S.M.A.R.T-2 preserves the compatibility with the WS-compatible clients, and discussed its promising interoperability with the existing Grids.

As the new system uses a peer-to-peer computing model, it exhibits significant different properties than the conventional server-based Grid systems. These differences reflect in organizational hierarchy, security and trust model, and connectivity characteristics. Future work of this research includes proposing a domain model for security and trust, and organizational hierarchy, and designing a network layer which solves the connectivity issues. Testing and evaluation statistics will also be provided in the near future.

References

Alfred, W. L. (2003), 'The future of peer-to-peer computing', *Communications of the ACM* **46**(9), 56–61.

Barkai, D. (2002), *P2P Computing*, Intel Press, Santa Clara, California.

Czajkowski, K., Foster, I., Karonis, N., Kesselman, C., Martin, S., Smith, W. & Tuecke., S. (1998), A resource management architecture for metacomputing systems, *in* 'Proceedings of IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing', Orlando, Florida, pp. 62–82.

Epema, D. H. J., Livny, M., vanDantzig, R., Evers, X. & Pruyne, J. (1996), 'A worldwide flock of condors: Load sharing among workstation clusters', *Future Generation Computer Systems* **12**(1), 53–65.

Fitzgerald, S., Foster, I., Kesselman, C., Laszewski, G. v., Smith, W. & Tuecke, S. (1997), A directory service for configuring high-performance distributed computations, *in* 'Proceedings of the 6th IEEE Symposium on High Performance Distributed Computing', IEEE Press, Portland, Oregon, pp. 365–375.

Foster, I. (2005), *A Globus Toolkit Primer*. http://www-unix.globus.org/toolkit/docs/4.0/key/GT4_Primer_0.6.pdf.

Foster, I., Czajkowski, K., Ferguson, D., Frey, J., Graham, S., Maguire, T., Snelling, D. & Tuecke, S. (2005), 'Modeling and managing state in distributed systems: the role of ogsi and wsrf', *Proceedings of the IEEE* **93**(3), 604–612.

Foster, I. & Kesselman, C. (1999), *The Grid: Blueprint for a New Computing Infrastructure*, 1st edn, Morgan Kauffman, San Francisco, California.

Foster, I., Kesselman, C., Nick, J. M. & Tuecke, S. (2002), *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration*. http://www.globus.org/research/papers/ogsa.pdf.

Globus Alliance (2005), *Globus Toolkit 4.0 (GT4)*. http://www-unix.globus.org/toolkit/docs/4.0/GT4Facts/.

Goldman, J., Rawles, P. & Mariga, J. (1999), *Client/Server Information Systems*, Wiley, Hoboken, New Jersey.

Lesser, V. (1999), 'Cooperative multiagent systems: A personal view of the state of the art', *IEEE Transactions on Knowledge and Data Engineering* **11**(1), 133–142.

Pfister, G. (1997), *In Search of Clusters*, 2nd edn, Prentice Hall.

Smarr, L. & Catlett, C. E. (1992), 'Metacomputing', *Communications of the ACM* **35**(6), 44–52.

Sun Microsystems Inc. (2004), *Java Object Serialization Specification*. http://java.sun.com/j2se/1.5/pdf/serial-1.5.0.pdf.

W3C (2002), *Web Services*. http://www.w3.org/2002/ws/.