2005

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

J. Tang  
*University of Wollongong*

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

Publication Details  
This article was originally published as: Tang, J & Zhang, M, An agent-based peer-to-peer grid computing architecture, First International Conference on Semantics, Knowledge and Grid (SKG ’05), Beijing, China, November 27-29 2005, 57. Copyright IEEE 2005.
An agent-based peer-to-peer grid computing architecture

Abstract
We propose a multi-agent based peer-to-peer grid computing architecture in this paper to solves the issues that the conventional grid architecture is limited in modeling computer systems with highly dynamic and autonomous computing resources, and its super-local resource management and scheduling strategy limits the utilization of the computing resources. Besides, the new architecture provides reasonable compatibility and interoperability with the conventional grid systems and clients.

Disciplines
Physical Sciences and Mathematics

Publication Details
This article was originally published as: Tang, J & Zhang, M, An agent-based peer-to-peer grid computing architecture, First International Conference on Semantics, Knowledge and Grid (SKG '05), Beijing, China, November 27-29 2005, 57. Copyright IEEE 2005.
An Agent-based Peer-to-Peer Grid Computing Architecture

Jia Tang and Minjie Zhang
School of Information Technology and Computer Science, University of Wollongong
Wollongong NSW 2522, AUSTRALIA
{jt989, minjie}@uow.edu.au

Abstract

We propose a multi-agent based peer-to-peer grid computing architecture in this paper to solve the issues that the conventional grid architecture is limited in modeling computer systems with highly dynamic and autonomous computing resources, and its super-local resource management and scheduling strategy limits the utilization of the computing resources. Besides, the new architecture provides reasonable compatibility and interoperability with the conventional grid systems and clients.

1 Introduction

The conventional computing grid has developed a service oriented computing architecture with a super-local resource management and scheduling strategy. 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 five issues of the conventional grid: 1) WSRF [1] 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; and 5) It is infeasible 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 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 grid. Section 3 probes into the compatibility and interoperability issues with the existing grid systems and clients, and concludes this paper.

2 Agent-based P2P Grid Architecture

In this section, we propose S.M.A.R.T-2 (Service-oriented, Microkernel, Agent-based, Rational, and Transparent), a multi-agent based peer-to-peer grid computing architecture with an adaptive resource management and scheduling strategy.

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. A microkernel grid container runs on every computing node. These containers serve as the runtime and managerial environment for the tasks. 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.

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 [4], and allows the clients to interact with the grid using SOAP messages.

Inside the container, there are four components: 1) the runtime environment (RT), provides the fundamental routines and the runtime libraries for both the agents and the modules; 2) the management agent, manages the container, the policies and the configurations; 3) the profiling agent, gathers the status of the network, the peers, and the running modules, and provides optimized dynamic configurations for the computing agent; and 4) 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 invocations among the peers while providing fault-tolerance and load balance.

S.M.A.R.T-2 uses modules to describe tasks. A module consists of the module description, the executables, the serialization, and the module owned files. The Module Description (MD) has two sections: the task section and the service section. The task section of MD defines the task related information, and consists of two subsections: 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; and 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.

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 to define the service interfaces. The executables are Java bytecode files or .NET executables. When a running module is suspended by a user or if it is relocated, it will be serialized. This process is equivalent to the object serialization [3] of Java. It allows the grid container to store the runtime dynamics of the module, and restore them when the execution of 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, and then uses the create method of the IModuleContext interface to create an instance of the startup module. Once the startup module is instantiated and runs, it can start instances of other modules by using the same interface.

In S.M.A.R.T-2, whenever a module is instantiated, 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 in order to release their occupied resources. 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).

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 least 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, and synapse for short) are the values. The four values are defined for a simulated synapse: 1) strength, whose range is [0, 1], represents the current strength of the connection. A value “1” means that the connection is a permanent connection; 2) \texttt{deathThreshold}, whose value is randomly selected from a user configured range, when a connection is created. When strength is less than \texttt{deathThreshold}, the connection is removed from the hash table, i.e. the connection breaks up; 3) \texttt{activateThreshold}. When a connection is created, a random value is selected from a user configured range as \texttt{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 \texttt{activateThreshold}, the connection becomes active, and the \texttt{activateThreshold} is set to 0; and 4) \texttt{permThreshold}. If an active connection’s strength continues growing to a value greater than \texttt{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. Various messages are generated by the peer which receives the client’s instruction, and then delivered to other peers before performing the any operation. These messages and the replied messages are encapsulated into impulses, and transmitted among the peers. This process is called relay.

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 transmitted 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 has 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.

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 purpose.

With the selection process, 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 that the grid will gain better connectivity and higher ratio of resource utilization, and work more efficiently.

3 Discussion and Conclusion

Recalling the task 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-T2.

In this paper, we firstly pointed out five issues for the grid in an autonomous, heterogeneous and highly dynamic environment. Aiming at these issues, we proposed our novel task model. With the serialization and the module owned files, the internal states of a task is easy to maintain, and the process is totally transparent to the users. We solve the service adaptability issue by spawning more service modules to accommodate new service requests. With the help of the relay process, the new task model provides a common programming interface that supports task decompositon, state persistence, and inter-task communications. We applied the peer-to-peer computing architecture to the grid, and innovatively proposed a multi-purpose message passing mechanism to achieve load balance, high ratio of resource utilization, and fault-tolerance. These mechanisms also allow the grid to intelligently reconstruct and utilize the computing resources.

References