A Peer-to-Peer Meta-Scheduler for Service-Oriented Grid Environments

Kay Dörnemann Jörg Prenzer Bernd Freisleben

Department of Mathematics and Computer Science, University of Marburg
Hans-Meerwein-Str. 3, D-35032 Marburg, Germany
{doernemk, prenzerj, freisleb}@informatik.uni-marburg.de

ABSTRACT

Meta-scheduling in a Grid is aimed at enabling the efficient sharing of computing resources managed by different local schedulers within a single organization or scattered across several administrative domains. Since current Grid meta-schedulers operate in a centralized fashion and thus are single points of failure, we present a distributed meta-scheduler for a service-oriented Grid environment based on peer-to-peer (P2P) networking techniques and ant colony optimization algorithms adapted to a P2P network. In the proposed approach, the meta-scheduling process provides automatic load balancing, is completely decentralized, fault tolerant, scalable, and does not require complex administration. Experimental results demonstrate that scheduling decisions are made quickly and lead to a good balance of the computational load.

Keywords

meta-scheduling, Grid computing, peer-to-peer computing, ant colonies

1. INTRODUCTION

Grid computing environments are heterogeneous collections of networked hard- and software components located at different sites and hosted by different organizations. To enable users to access these resources in a convenient manner using standardized interfaces, service-oriented Grid middleware systems based on the Web Services Resource Framework (WSRF) such as Globus Toolkit 4.x [2], gLite [1], or Unicore/GS [3] have been developed. Typically, they provide functionality for runtime components, execution management, information management, data handling and security.

The focus of this paper is on job scheduling which is part of execution management. Jobs are computational tasks described by a file (XML in general). They may perform input/output operations while running, which affects the state of the computational resources and their associated file systems. Jobs are submitted via a Grid service interface offered by the mentioned Grid middleware systems to translate the job description into a specific format accessible by the installed local schedulers, such as Sun Grid Engine, Torque/PBS, Condor, LSF or even *fork* as the basic operating system scheduler for single computers. The local schedulers are responsible for selecting the local Grid resources to execute a job.

Consequently, in a large Grid there may be several Grid sites running those middlewares, each of them with an interface to one or more local schedulers responsible for a compute cluster or just a single computer. In this scenario, it is necessary to decide to which Grid site a job should be submitted. For this purpose, global knowledge about the Grid environment is needed. Apart from static information, such as available hard- and software at the site, dynamic up-to-date information, such as CPU utilization, free memory, free hard disc space etc. needs to be considered.

Meta-schedulers have been proposed to manage the various types of information to make such a decision. They provide a standardized interface to all service-oriented Grid middleware site-schedulers combined with a decision process that enables all Grid users to submit a job with special requirements at a single point and without any global knowledge. Normally, information about the local schedulers must be manually configured at the meta-scheduler. Since in gneral there is only a single meta-scheduler for each Grid, the meta-scheduler is a single point of failure and could also become a performance bottleneck. Our idea to avoid these drawbacks is to build a distributed meta-scheduler based on peer-to-peer (P2P) overlay networking technologies. P2P networks are usually self-configuring, self-repairing and support highly dynamic changes in the network configuration. Our particular focus is on P2P networks based on distributed hashtable (DHT) solutions, because they guarantee that all nodes can be found in the P2P space and all nodes can be identified by a unique ID. Clearly, an application layer overlay network, as built by P2P solutions, comes along with a higher inefficiency because it implements functionality that is already provided by lower layers (like routing and failure handling). Additional layer(s) add overhead to each transmitted message and reduce the effective payload. However, P2P networks have advantages that outweigh the mentioned disadvantages, such as self-configuring and self-repairing functionalities, predefined message formats and special routing algorithms for search and other purposes. ıtions are easier to implement using a

GridNets 2007 October 17-19, 2007, Lyon, France.
Copyright 2007 ICST ISBN 978-963-9799-07-3.
DOI 10.4108/gridnets.2007.2164

P2P framework and can interact in a closed network.

Thus, in this paper we present a P2P meta-scheduler that combines service-oriented Grid technology (in our implementation: Globus Toolkit 4) with distributed algorithms that operate in a P2P environment. The overall goal is to construct a system in which users can submit jobs to any node in order to trigger the meta-scheduling process which will eventually find the most appropriate node for job execution in the system and also performs automatic load balancing. This meta-scheduling process is completely decentralized, fault tolerant (jobs must not be lost because of node failure or network errors), scalable, dynamic and does not require complex administration. It is based on state-of-the-art P2P techniques for the meta-scheduler and ant colony optimization (ACO) algorithms adapted to a P2P network. We have chosen the ACO algorithms because they are quite simple, easy to implement and fully decentralized. Although security is an important topic especially in Grid computing, we explicitly do not discuss security issues here, since a forthcoming paper will be devoted to these issues. Finally, experimental results will be presented to demonstrate that scheduling decisions are made quickly and lead to a good balance of the computational load.

The paper is structured as follows. Section 2 presents the architecture of the proposed approach. Section 3 describes the ant colony algorithms used in our solution. Experimental results are presented in Section 4. Section 5 discusses related work. Section 6 concludes the paper and outlines areas for future research.

2. A P2P META-SCHEDULER

The architecture of the proposed P2P meta-scheduler is presented in Fig. 1.

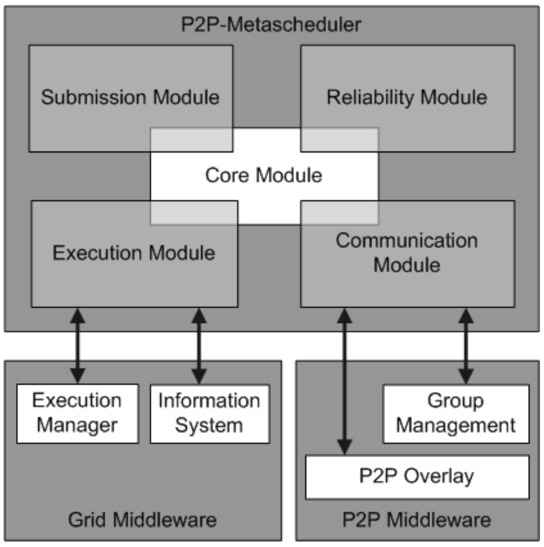

Figure 1: Architectural Overview

All components shown in Fig. 1 are available at each computing node, which hosts one P2P and one Grid node (as entities of the corresponding middleware). The Core Module is the heart of the P2P meta-scheduler and therefore provides the basic functionality, such as job management, scheduling algorithms and communication between the other components. The other modules gather around the Core Module to provide supportive functionality and the interfaces to the Grid and P2P middlewares. The Submission Module's function is to completely handle the interaction with users or other meta-schedulers. These can submit, cancel or request the status of jobs that are supposed to be scheduled. The Execution Module submits, monitors and handles job execution in close interaction with the service-oriented middleware scheduler. It is also responsible for the interaction with a Grid information space that is used to provide vital information for the meta-scheduling process. For this kind of interaction, it uses the service-oriented interface provided by the Grid middleware. The Communication Module represents the main interface to the underlying P2P network and manages all the related information, such as IDs, group memberships and so on. The Reliability Module is concerned with fault tolerance with respect to job scheduling, transportation and execution. Failure safety is desired because lost, failed or duplicated jobs have to be avoided.

Jobs can generally be regarded as job descriptions in combination with optional values for a deadline (a date when the job has to be finished), duration (a runtime estimation for the job), maximum cost and a set of requirements that have to be fulfilled by all nodes that can be taken into account for job execution. Possible requirements are, for example, CPU speed, free hard disk space, available memory, etc.

The protocols for exchanging information between instances of our architecture residing on each node are based on the Anthill framework [6] which provides a system where interconnected `AntNests` offer services to a user based on the work of autonomous agents, called `Ants`. Typically, a peer node has exactly one running `AntNest` which receives, processes, schedules and sends `Ants` to neighbor `AntNests` over a communication layer and manages local resources using middleware interfaces.

Ant colony optimization (ACO) algorithms have emerged in the 1990s and represent a class of algorithms that were initially used to solve (NP-hard) optimization problems based on a metapher inspired by the collective problem solving behaviour of natural ants. Contemporary ACO algorithms provide solutions for the traveling salesman problem, routing problems, assignment problems, machine learning problems, scheduling problems and several others [7]. All solutions rely on the basic ideas derived from real-life ants. A single ant operates autonomously (in cases of labor division, transport etc.), but as a whole they provide highly coordinated behavior in order to solve complicated problems without using any centralized control [7]. Ants generally coordinate their actions by using so-called pheromones which can be set by ants during a decision process and have influence on the decision-making of following ants. Every time an ant has to make a decision during an algorithm (e.g. which way to go next), it sets a pheromone mark according to the decision outcome (e.g. way A has been chosen). The following ants are able to read the pheromone marks of previous ants and make their decision by considering the different pheromone strengths for every possible decision outcome. Because ants that make the right decisions will solve the problem in a shorter period of time, the pheromone marks of the best option will grow a lot faster than the others and the following ants are pushed towards the globally optimal solution. The algorithms used in our P2P meta-scheduler are based on this idea.

In our approach, Ants can be regarded as autonomic agents that carry information about other nodes, transport jobs and trigger certain events in AntNests. The network exploration of our Ants is not deterministic. Their decision about which AntNest to target next is made by considering the current AntState. The list of already visited nodes (new nodes are preferred) and generally all the information about neighboring nodes is present in the current AntNest's routing table (see section 3.1). The Core Module is responsible for handling the Ants and can be regarded as the AntNest. Receiving and sending out the Ants is the task of the Communication Module, which stands in direct contact with the underlying P2P network. In the ACO context, the Communication Module can be regarded as the entrance to the AntNet. The other modules are not part of the AntNest but nevertheless important for the system. For communication purposes with other meta-schedulers or users, the Submission Module receives the jobs to dispatch and hands them over to the Core Module. The Reliability Module uses the peer grouping mechanism of the underlying P2P network for reliability purposes. The connector to the Grid middleware is the Execution Module. The functionality of these components will be described in more detail later.

In our P2P meta-scheduler, Ants can operate in three different states that determine their behavior. The Free state is the initial state that Ants are in after their creation. In this state, Ants just wander around as long as their time-to-live value (TTL) allows, spread information about other nodes to the visited AntNests and look for jobs that can be picked up for scheduling. If a job is found, the Ant picks it up and switches to the SearchPeer state in which it searches for an appropriate node for job execution. After having visited enough nodes in order to make a scheduling decision, the Ant switches to the DirectTransfer state and transfers the job to the chosen node where it is scheduled using the local scheduler. For single Grid nodes this local scheduler may be *fork* or even no scheduler. In the case of a Grid node that represents a cluster, Sun Grid Engine, Torque or other schedulers are used. Having deposited the job, the Ant returns to the Free state, as shown in Fig. 2.

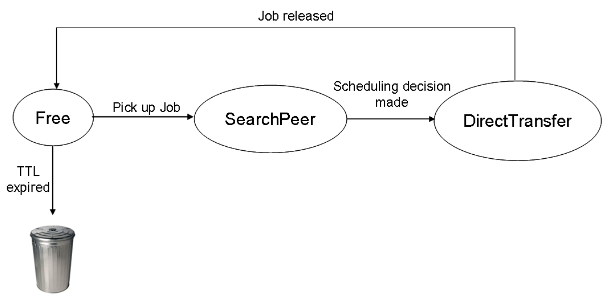

Figure 2: Ant states.

3. ANT ALGORITHMS

3.1 Ant Routing and Forwarding

Ants are routed through the P2P network according to the information that is made available by routing tables which are located at every AntNest (and every Ant as well). These routing tables have one entry for every remote node that is known by the AntNest. Ants carry routing tables in order to spread the newest entries for all nodes throughout the P2P network. Each entry in the AntNest routing table consists of the following components: NodeId, localTimestamp, loadValue, fitRatio, QueueLength, prefValue, Grid$. The first two components identify the node which has published this information about itself, and the local timestamp (the clock value of the specific node) indicates when the information has been generated. The rest of the entry consists of pheromone components, which form the pheromone trail that is used to forward Ants. loadValue describes the local load (CPU usage) of the specific node whereas fitRatio indicates the general usability of a node regarding job execution. Since a job can be equipped with special requirements that must be fulfilled by a node in order to execute the job, fitRatio is defined as the quotient (number of processed Ants that carried a job whose requirements matched the nodes properties)/(number of all processed Ants in SearchPeer state). QueueLength is the length of the queue which contains jobs that are ready to be carried away by the next Ant. Grid$ describes the cost of one computational unit (e.g. a CPU second) that the user who submitted the job has to pay if the job will be executed at the specific node. This all takes place in the Core Module.

As already mentioned, Ants carry routing tables to spread information. When an Ant arrives at an AntNest, the Communication Module receives its routing table and hands it over to the Core Module where both routing tables are merged such that both tables contain the same information for every node. If an entry for a specific node is present in both tables, the entry with the higher localTimestamp will be chosen to prevail because it is newer. By performing this *two-way-merge*, Ant and AntNest get the newest information about nodes that are currently in the P2P network.

The routing tables play an important role in the process of determining the next node for an Ant. When the decision has to be made, a subset of the "pheromone components" is used to calculate a single weight value for each entry in the AntNest routing table. The calculation of this value depends on the current state of the Ant. In the Free state, QueueLength increases the weight (see 3.5), while the fact that an Ant has already visited the current node has a decreasing effect. In the SearchPeer state, loadValue decreases the weight, whereas fitRatio increases it. In this case, previously visited nodes are disadvantaged. Grid$ is a knock-out criterion if an explicit maximum cost value is assigned to the carried job. In both cases, prefValue has an increasing effect on the calculated weight value.

The calculated weight value for every node in the routing table is now used as input for a function which chooses the desired node randomly while preferring the nodes with a comparably higher weight. This corresponds to the behavior of real Ants that walk randomly but are influenced by pheromone trails.

3.2 Ant Population and Congestion Control

A major source of problems in our approach might result from the unequal distribution of Ants in the P2P network. While one node might be overwhelmed by arriving Ants and not be able to process them in a reasonable period of time, another one might not have enough Ants in order to schedule jobs or to publish the local routing table entry. The previously described routing algorithm makes this even worse because certain nodes (with the highest performance capabilities) are preferred to others. Since in a dynamic distributed system no global information is present at a single location, a set of algorithms and mechanisms is necessary to solve this problem.

In a stable system (with no node failures and departures), keeping the overall number of `Ants` constant is simple. In a dynamic system, however, node failures, node departures or the already mentioned unequal `Ant` distribution might lead to a regional insufficiency of `Ants`. A straightforward solution is to allow every `AntNest` to create and remove `Ants`.

However, handling regional and overall overpopulations requires more sophisticated measures. Several known algorithms have been developed to address this issue [20], but they all can lead to mass mortality of `Ants`, require a synchronous system or try to compulsively maintain a constant inter-arrival time at every `AntNest`. In the given context, all of this is neither desirable nor necessary. The algorithm we propose is based on a constant monitoring of the `Ant` processing latency (i.e. the difference in time between arrival and departure of an `Ant`) at every single `AntNest`. We use adaptive blacklists to prevent sending `Ants` to unavailable nodes and to shut down the reception of `Ants`.

The fact that `Ants` have a global unique time-to-live (TTL) value allows every `AntNest` to create a new `Ant` after the TTL-th `Ant` has been processed. Therefore, it makes sense to define a global maximum latency that a single `AntNest` is willing to wait since the last arrival of an `Ant`. If this maximum amount of time has passed for a specific `AntNest`, it simply creates a new `Ant` that is processed immediately. If the monitored `Ant` processing latency exceeds a globally defined value for the last `Ant` in the queue, then two countermeasures can be utilized. The first one is to shut down the reception of any new `Ants`. This measure utilizes the fact that neighboring `AntNests` then regard the "shut down" node as temporarily unavailable, and the `Ants` put it on their adaptive blacklists. Adaptive blacklists prevent an `AntNest` from sending `Ants` to nodes which have not been available recently. However, a node can also disappear from a blacklist when the `AntNest` receives a newly propagated routing table entry from an arriving `Ant`. The whole concept of adaptive blacklists is described in section 3.3. By following this procedure, the local queue of `Ants` can be relieved and processed before new `Ants` arrive. If the monitored threshold is not exceeded anymore, the reception of new `Ants` can be turned on again.

The described solution does not provide any measures for handling global or regional overpopulation of `Ants`. For this reason, a second countermeasure has to be utilized in the following way. If the monitored threshold is exceeded significantly, then, in addition to shutting down `Ant` reception, `Ants` in the local queue that are in their `Free` state can simply be deleted in order to relieve the queue. The return to a normal state can be performed as described above. All of this is handled by the Core Module.

3.3 Failure Safety

In a dynamic distributed P2P system, a large variety of failures can occur. The most common ones that are handled in our approach are failures of the communication subsystem (temporary or permanent network failure) and node failures. Failures generally have an impact on two major functions of the P2P meta-scheduler. The first one is the direct transfer operation of an `Ant` between two `AntNests` and the second one is the safe and secure scheduling and processing of jobs. The transfer of an `Ant` between two nodes has to be made fail-safe since the underlying P2P network does not provide any functionality in that direction. The fail-safe transfer of `Ants` between two nodes can be provided by means that are common in transport-level protocols like TCP. These measures include message IDs, timeouts, resends, replies etc. Since they already have been sufficiently discussed in the literature, there is no need for a detailed description here.

More interesting is the impact of failures on the scheduling of jobs. We have discovered two major failure scenarios which can occur in different stages of the scheduling process. The first scenario is the crash of a node while an `Ant` is visiting, i.e. the `Ant` and jobs that it carries get lost. Moreover, the currently executed jobs on the failing node will also be destroyed. Here, the P2P meta-scheduler has to define a decentralized mechanism for detecting the loss of jobs in their scheduling or execution stage and for recovering from these losses by re-inserting those jobs into the system.

The second failure scenario is caused by temporary or permanent failures in the communication subsystem (if more than one node is concerned, we have a network partitioning). This means that a node is not reachable by other nodes anymore, but internally is still functioning (the separated node might still be able to execute the local jobs). For a permanent failure, this scenario can be handled exactly like the first scenario. The case of a temporary network partition is more complicated because it is not possible to distinguish between a permanent and temporary failure at the moment when the decision for re-scheduling has to be made. This may result in duplicated jobs and must be avoided. Our general solution in this case is to allow job duplications as long as the network is partitioned, but to return to a consistent state (i.e. without duplicated jobs) when the separated networks are eventually merged again. Our algorithm proposed to solve this problem is described in the following.

A user submits a job to the P2P meta-scheduler's Submission Module, which hands it over to the Core Module where it gets transfered to the Communication Module and from there to the node in the P2P network. When a job is submitted at any node, a globally unique identifier (UUID) for the job is generated, which is used by the Reliability Module to construct a unique `multicast group` (in the following: `JobTrackerGroup`) in the P2P network to track and monitor the location and execution status of the dedicated job from the creation to the successful or unsuccessful completion of execution. To construct the `JobTrackerGroup` for a newly submitted job, the `AntNest` that received the job description sends a join-message to a set of known nodes (the size of the set is predefined) and asks them to join the multicast group for the job with identifier UUID. When all the requested nodes have joined the `JobTrackerGroup`, the job is ready for handling by an `Ant`. In the following, the `Ant` that carries the job sends an alive message to the `JobTrackerGroup` every time it arrives at a new `AntNest`. When the scheduling process has succeeded and the job is executed by a particular node, periodic alive messages are sent to the `JobTrackerGroup`.

Every member of the `JobTrackerGroup` continuously checks for timeouts, i.e. for a predefined time no alive-messages have been received. When a timeout is detected, the corresponding node asks the other members whether they agree to the timeout or not. If they do not agree (false alert, possibly caused by message loss), they all continue as if nothing has happened. If they do agree, all members know that the implied job has timed out. The task now is to determine a coordinator, i.e. a node that performs the re-scheduling. The necessity for having a coordinator arises due to the possibility that one or more members have departed during the time since the `JobTrackerGroup` has been created. There are several existing distributed election algorithms [14] [15] which can be utilized to solve this problem. One quite simple mechanism is to assign to every member of the `JobTrackerGroup` a unique delay value that the node has to wait between the detection of a timeout and the actual re-scheduling of the job. The coordinator node does two things after the agreement over the timeout has been approved:

- First, it sends a message to all other members to inform them of the upcoming re-scheduling (in the simple election mechanism, the receivers cancel their candidature as coordinator) and checks if the `JobTrackerGroup` still has sufficient members. If not, new members are invited to participate.
- Second, the job is re-scheduled and the `JobTrackerGroup` restarts the job monitoring and tracking.

The described modus operandi, however, does not yet solve the problem of duplicated jobs caused by a temporary network partition. For this purpose, the self-repairing features of the peer grouping mechanism can be used if provided by the P2P network. When the formerly partitioned parts of the network join again, the grouping mechanism automatically repairs the multicast group and all members of the `JobTrackerGroup` will receive multiple alive-messages from different nodes caused by the duplicate jobs. This fact can be detected by the group, and one of the duplicate jobs can be canceled in order to rebuild the consistent state. Finally, when job execution completes, a corresponding message is sent to the `JobTrackerGroup`, so every member can stop checking for timeouts.

3.4 Node Evaluation

During the participation of a node in the Grid and the P2P meta-scheduler, the node's individual performance properties become known and can be used to improve meta-scheduling. The main idea behind this node evaluation is to route job-carrying `Ants` preferably to nodes which have proven to be reliable and powerful in the past. For this reason, the routing table attribute `prefValue`, which has already been described above, can be used to influence the routing decision in the desired way. At first, it is necessary to find appropriate criteria to determine the `prefValue` for each node. In this context, the `prefValue` is preferably calculated by the quotient (number of jobs that could be completed successfully before the deadline)/(number of all locally executed jobs) for each node. In addition, a considerably smaller actual duration of the job execution than estimated by the user might also have a positive influence on the `prefValue`. Every node has to calculate that value continuously and update its routing table entry accordingly.

3.5 Load Balancing

Apart from matching Job requirements to node properties, load balancing is an important criterion for successful meta-scheduling. Appropriate load balancing algorithms need to operate non-preemptively in this context, because it is not (yet) possible to migrate running jobs from one node to another node by current Grid middleware systems. Several distributed load balancing algorithms are based on the principle of collecting sufficient information locally and making the scheduling decision at a single node (mostly the one where the job is located) [16] [5]. Other algorithms are based on autonomous agents that migrate between nodes, collect specific information needed for job scheduling and eventually decide where the job will finally be executed [21] [18]. Clearly, this category of algorithms is very well-suited for the existing agent-based infrastructure of our P2P meta-scheduler. The Messor Ant Load Balancing algorithm [18] can be easily fitted into the existing routing mechanism, as described in the following.

In the original algorithm, `Ants` operate in two phases. In the first phase, an `Ant` searches for highly loaded nodes (`searchMax`). In the P2P meta-scheduler, this `searchMax` state of the original algorithm corresponds to the `Free` state, in which `Ants` are preferably routed towards nodes with a long queue of jobs that can be scheduled. When an Ant in its `Free` state reaches a node with a non-empty job queue, it picks up the job and switches to the `searchPeer` state (which corresponds to the `searchMin` state of the second phase of the Messor Ant algorithm). In this state, `Ants` are preferably routed towards nodes that have a low local load until a predefined threshold for (`minLoad/maxLoad`) is reached (the values for `minLoad` and `maxLoad` are the minimum and maximum load values of previously visited nodes). Then, the visited node that is able to execute the job and has the lowest load value will be selected to execute the job. As an alternative to load balancing, `Ants` can also operate in an *Economic Grid* environment [9] [11]. With some small changes to the previously described algorithm, Ants can explore the P2P network and select nodes which offer the least cost for job execution (simply select the visited node with the lowest `Grid$-value`) or nodes which are expected to give the best performance while charging less than a maximum price (for performance comparisons `prefValue` and `loadValue` is considered, for cost `Grid$`).

The algorithm is illustrated in Fig. 3. The upper picture shows the normal order of events:

1. A client submits a job to `AntNest` B. A new `JobTrackerGroup` is created (`AntNests` B and A); each of its members holds the complete job information.
2. `Ants` are preferably routed towards `AntNest` B, because the job queue is not empty. An Ant picks up the job and searches for an `AntNest` that fits to the requirements and is appropriate according to the other scheduling criteria. At every visited `AntNest`, an additional message is sent to the `JobTrackerGroup` to indicate the current Ant position.
3. The job is released by the `Ant` and `AntNest` F starts to execute the job. `Alive-Messages` are sent out periodically to inform the `JobTrackerGroup` about the current status of the job. When the job is finished, the `JobTrackerGroup` is informed.

The lower picture shows the order of events in the case of a node failure:

1. `AntNest` F fails during job execution and is not available any longer.

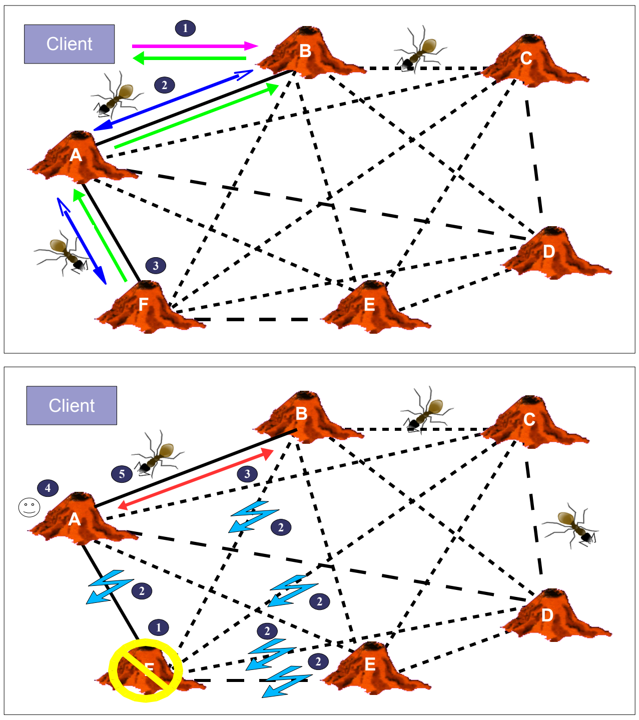

Figure 3: Load balancing example

2. Because Ants cannot be sent to `AntNest` F any more, other `AntNests` put F on their adaptive blacklist (this has nothing to do with job execution). Furthermore, `AntNests` A and B do not receive `Alive-Messages` any more and realize that the job is lost.

3. `AntNests` A and B start a distributed election to decide which one of them is allowed to re-insert the job.

4. `AntNest` A is elected and puts the job into its queue.

5. An Ant passes `AntNest` A in the `SearchPeer` state and picks up the job. The scheduling process starts again.

4. EXPERIMENTAL RESULTS

The prototypical implementation of our approach is based on the *FreePastry* P2P framework [19]. FreePastry is an open source Java implementation and provides a peer-grouping application called Scribe which allows to arrange peers in a self-repairing multicast tree.

The service-oriented Grid middleware used in our implementation is Globus Toolkit 4.0.3. Its scheduler interface is called WS-GRAM (Web Service Grid Resource Allocation Manager) which provides a Grid service interface to one or more local schedulers. The Monitoring and Discovery System (MDS 4) is the Globus information space which stores information from several sources as a XML-document and is accessed through Grid service interfaces.

Our P2P meta-scheduler has been implemented in Java (1.5/1.6). Job requirements are XPath statements which can be evaluated to a boolean value when used with the XML-compliant information provided by the local MDS 4.

The following evaluation demonstrates the feasibility of our approach and shows its performance in a set of experiments.

4.1 Test Environment

Our test environment are 10 workstations with Intel Pentium 4 and Intel Core Duo CPUs, one or two gigabytes of main memory, 100 MBit network interfaces and a switched network. The used Java virtual machine is version 1.6.0.

4.2 Test Cases

The goal of our test series is to obtain information about the speed in which the P2P meta-scheduler reaches certain quality levels concerning the fulfillment of given scheduling criteria. The selected scheduling criterion is the load balancing between the participating nodes. We assume that all jobs have the same load effect, which allows us to express the quality of load balancing as the standard deviation of the number of jobs: the lower the standard deviation, the better the load balancing.

The time required for job distribution, as expressed by the number of hops reclined by a job carrying ant until a scheduling decision is made, is one of the parameters we measured. Another measured parameter is the total time required from job delivery to the completion of the allocation. The corresponding threshold (`minLoad/maxLoad` < threshold) for the scheduling process described in section 2 is set to 0.95, which implies a fast job distribution.

In total, 12 measurements with 11, 21, and 31 nodes respectively AntNests running on the workstations were performed. The unusual number of nodes is explained by a boostrap node for the P2P network and an even number of peers. Different experiments in which 250, 500, 1000 and 2000 jobs were initially submitted to a single AntNest were conducted to evaluate the load balancing capabilities of our approach.

4.3 Results

The results of our experiments are shown in Fig. 4. The standard deviation is lower than five jobs, which means that the scheduling criterion is fulfilled with high quality. The low total time for the scheduling process (50 seconds for 2000 jobs and 11 AntNests) indicates that our solution is fast enough for real world applications. The efficiency of our routing algorithm is expressed by the few hops the ants needed to schedule a job.

The scheduling works better, the more unbalanced the load at the beginning, the more dynamic the system and the more heterogeneous the load effect of the jobs is. The test cases used here had opposite characteristics and can be seen as worst cases. The algorithm tries to find a node that fits best to the job requirements instead of placing it at the next node. This effect results in a higher number of hops. In a real world Grid application, the results may be even better than in our worst case scenario.

5. RELATED WORK

Buyya et al. [10] present some real world economic models such as an auction model, bargaining model, commodity market model, contract-net model and bartering model. The authors reason how these economic models can used in Grid computing environments, however, without presenting an implementation. Economic Grid scheduling is an interesting

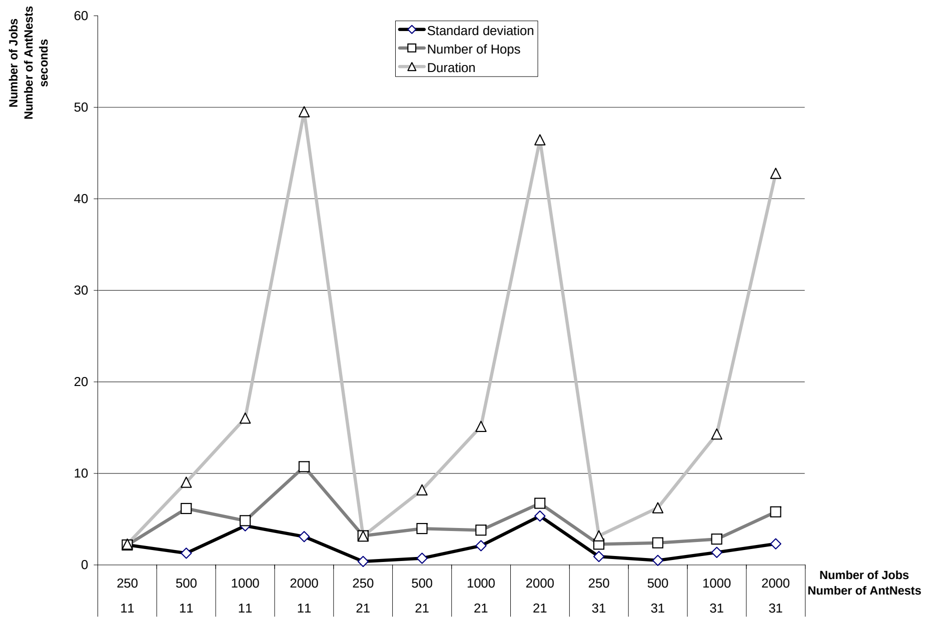

Figure 4: The first row below the X axis shows the number of jobs to distribute and the second row shows the number of AntNests for the particular test. The Y axis shows the standard deviation as the number of jobs, the number of hops as the number of AntNests and the total time in seconds from job delivery to the completion of the allocation

subject which can be realized as an add-on to our P2P meta-scheduler implementation.

Nimrod/G [8] is a Grid aware (the implementation is based on the Globus Toolkit) resource management and scheduling system based on the parameter sweeping system Nimrod [4]. The architecture is extensible to use any other Grid middleware services such as Legion, Condor and NetSolve. Nimrod/G comprises a centralized architecture and makes an attempt to incorporate economic ideas into scheduling. Each resource is associated with a price and each job has a given budget. Nimrod/G focuses on managing the execution of parametric studies. Our P2P meta-scheduler is not devoted to a special type of jobs. Due to its decentralized architecture, it is more fail-safe than Nimrod/G.

GridWay is a centralized meta-scheduler developed by Huedo et al. [17]. GridWay enables large-scale, reliable and efficient sharing of computing resources managed by different local schedulers within a single organization (enterprise Grid) or scattered across several administrative domains (partner or supply-chain Grid). In contrast to our P2P meta-scheduler, GridWay is a single point of failure in a Grid.

Chakravarti et al. [12] present an agent-based approach to perform a distributed Cannon matrix multiplication in an unstructured, large-scale P2P network. A tree-structured overlay network is constructed in which a receiver-initiated dynamic scheduling process is initiated. Fault tolerance is preserved under utilization of the tree structure, i.e. parent nodes monitor their children and take measures in case of failures and vice versa. The main difference to our approach is the fact that we have developed a sophisticated Ant routing and forwarding scheme which can be used regardless of the desired computation. Moreover, our approach provides seamless integration into contemporary service-oriented Grid environments, such as the Globus Toolkit 4.x.

In [13], Chakravarti et al. propose a new design for desktop Grids that relies on a self-organizing, fully decentralized approach called the Organic Grid which is modeled according to the way complex biological systems organize themselves. Existing Grid middleware systems are not supported, and the execution of jobs is handled by the developed system itself instead of using local schedulers. For a compute cluster site, the software must be installed on each cluster node which usually is a no-go. In contrast, our solution is capable of handling single Grid nodes or compute clusters. Since we have developed a meta-scheduler, we do not have to care about the local scheduling environment.

6. CONCLUSIONS

In this paper, we have presented the design and implementation of a P2P meta-scheduler that is able to decide where to execute a job in a service-oriented Grid environment consisting of several administrative domains each running different local schedulers. Our approach is completely decentralized, fault tolerant, scalable, dynamic and does not require complex administration. It is based on the ant colony metapher: Ants act as autonomous agents and AntNests offer information services to support the decision process. This information is stored in routing tables, which are dynamically updated whenever an Ant arrives at a node or AntNest. We presented a method to equally distribute Ants in a dynamic distributed system by constantly monitoring the Ants' processing latency at every AntNest. Furthermore, an approach to handle failures of the communication subsystem and node failures to assure the safe and secure scheduling of jobs has been proposed. Experimental results have shown that scheduling decisions are made in a short amount of time and lead to a good balance of the computational load.

There are several topics for future research. For special purposes, additional scheduling criteria might be of interest (in Data Grids, for example, it is important to execute a job on a node which has a minimum cost for accessing the required data repositories) which need to be represented as pheromone components. Furthermore, an adequate scheme has to be developed to give every new pheromone component a specific weight compared to the other pheromone components so that Ants can be routed accordingly considering their current state. For population control, more research (stochastically or empirically) is required regarding the relationship between the number of Ants in a dynamic Grid environment and the corresponding scheduling performance in order to develop more sophisticated methods for performance optimization. In this context, it is also interesting to control the distribution of Ants more actively such that the performance of the P2P meta-scheduler can be maximized. For economic scheduling, it needs to be determined which market models (existing or new ones) will prevail in practice and how they can be efficiently integrated into the P2P meta-scheduler. Another goal is to achieve a scheduling process for Grid services which allows to implement computational tasks as Grid services and schedule them like jobs as treated in this paper. Scheduling Grid services is mainly important for on-demand Grid environments where each node runs the Grid middleware, but "classic" Grids in which each Grid middleware node represents a compute cluster could also benefit. Another topic for future research could be replacing the ACO algorithms by other algorithms and comparing the results.

7. ACKNOWLEDGEMENTS

This work is financially supported by the German Federal Ministry of Education and Research (BMBF) (D-Grid Initiative).

8. REFERENCES

[1] EGEE – Enabling Grids for E-Science, http://www.eu-egee.org.

[2] The Globus Alliance, http://www.globus.org.

[3] The Unicore Forum, http://www.unicore.org.

[4] D. Abramson, R. Sosic, J. Giddy, and B. Hall. Nimrod: a tool for performing parametrised simulations using distributed workstations. *High-Performance Distributed Computing*, pages 112–121, 1995.

[5] W. Aiello, B. Awerbuch, B. M. Maggs, and S. Rao. Approximate load balancing on dynamic and asynchronous networks. In *Symposium on Theory of Computing*, pages 632–641, 1993.

[6] O. Babaoglu, H. Meling, and A. Montresor. Anthill: A framework for the development of agent-based peer-to-peer systems. In *22th International Conference on Distributed Computing Systems*, pages 15–22, Vienna, Austria, 2002. IEEE.

[7] C. Blum. Review of "ant colony optimization" by M. Dorigo, T. Stützle, MIT Press, Cambridge, MA, 2004. *Artificial Intelligence*, 165(2):261–264, 2005.

[8] R. Buyya, D. Abramson, and J. Giddy. Nimrod/G: An architecture of a resource management and scheduling system in a global computational grid. *Computing Research Repository*, cs.DC/0009021:1–7, 2000.

[9] R. Buyya, D. Abramson, and J. Giddy. A case for economy grid architecture for service-oriented grid computing. In *IEEE International Parallel and Distributed Processing Symposium*, pages 83–98. IEEE Computer Society, 2001.

[10] R. Buyya, D. Abramson, J. Giddy, and H. Stockinger. Economic models for resource management and scheduling in grid computing. *Concurrency and Computation: Practice and Experience*, 14:1507–1542, 2002.

[11] R. Buyya, H. Stockinger, J. Giddy, and D. Abramson. Economic models for management of resources in peer-to-peer and grid computing. In *Proceedings of Society of Photo-Optical Instrumentation Engineers Vol. 4528, p. 13-25, Commercial Applications for High-Performance Computing*, 2001.

[12] A. J. Chakravarti, G. Baumgartner, and M. Lauria. Application-specific scheduling for the organic grid. In *IEEE International Conference on Cluster Computing, 2004*, pages 146–155. IEEE Computer Society, 2004.

[13] A. J. Chakravarti, G. Baumgartner, and M. Lauria. The organic grid: Self-organizing computation on a peer-to-peer network. *IEEE Transactions on Systems, Man, and Cybernetics*, 35:373–384, 2005.

[14] Chang and Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22:281–283, 1979.

[15] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, C-31(1):48–59, 1982.

[16] B. Ghosh and S. Muthukrishnan. Dynamic load balancing in parallel and distributed networks by random matchings. In *Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 226–235, 1994.

[17] E. Huedo, R. S. Montero, and I. M. Llorente. A framework for adaptive execution in grids. *Software, Practice and Experience*, 34(7):631–651, 2004.

[18] A. Montresor, H. Meling, and Ö. Babaoglu. Messor: Load-balancing through a swarm of autonomous agents. In G. Moro and M. Koubarakis, editors, *International Workshop on Agents and Peer-to-Peer Computing*, volume 2530 of *Lecture Notes in Computer Science, Springer*, pages 125–137, 2002.

[19] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science, Springer*, 2218:329–351, 2001.

[20] T. Suzuki, T. Izumi, F. Ooshita, and T. Masuzawa. Biologically inspired self-adaptation of mobile agent population. In *DEXA Workshops*, pages 170–174. IEEE Computer Society, 2005.

[21] Y. Wang and J. Liu. Macroscopic model of agent-based load balancing on grids. In *Autonomous Agents and Multiagent Systems*, pages 804–811. ACM, 2003.