# Decentralized As-Soon-As-Possible Grid Scheduling: a Feasibility Study

Xenofon Vasilakos
Athens University of Economics and Business, Greece
Email: xvas@aueb.gr

Jan Sacha
VU University Amsterdam, The Netherlands
Email: jsacha@cs.vu.nl

Guillaume Pierre
VU University Amsterdam, The Netherlands
Email: gpierre@cs.vu.nl

***Abstract*—Grid systems tend to grow in size, but currently deployed state-of-the-art schedulers have inherent scalability limits due to centralization and high messaging cost. In this paper, we explore the feasibility of scalable grid scheduling using a peer-to-peer overlay. We propose $DGS_{ASAP}$, a decentralized scheduling algorithm that schedules compute-intensive jobs such that their execution starts as soon as possible. Simulations of a 5000-node grid show that our design can scale to a large number of nodes, maintaining high grid utilization.**

***Index Terms*—grid computing, scheduling, decentralized, peer-to-peer**

## I. Introduction

The scale and complexity of Grid systems constantly increases. The largest currently Grid infrastructures, such as the LHC Computing Grid [1], consist of hundreds of resource centers located world-wide providing tens of thousands of CPUs. It is expected that future Grids will achieve an even greater scale. As the number of sites, nodes, and users in the Grid grows, efficient job scheduling becomes increasingly difficult and traditional approaches to job scheduling gradually become inadequate. Centralized schedulers, often used in single clusters, do not scale and introduce policy problems since they require a single authority to coordinate the resources of the entire Grid. Hierarchical schedulers, more commonly used in larger Grids, only partially address the challenge since they rely on metaschedulers which also have scalability and reliability limits. We believe that fully decentralized designs will be needed to cope with the scale of next-generation Grids.

In this paper, we study the feasibility of a decentralized and scalable Grid scheduler based on a peer-to-peer (P2P) overlay, a work that was done within the context of the XtreemOS project [2]. In contrast to previous applications of P2P overlays to Grids, we investigate the problem of job scheduling rather than mere resource discovery. Specifically, we propose $DGS_{ASAP}$: a scheduling model where compute-intensive jobs are executed on resource nodes *as-soon-as-possible (ASAP)*. Our approach is based on two main principles. First, each resource node manages its own schedule and autonomously decides on the jobs it is going to execute. There are no dedicated servers or coordinators which maintain a global schedule for the entire Grid or a site in the Grid. Second, a job can be submitted to any of the resource nodes. There are no predefined servers which schedule jobs on behalf of users. Instead, all resource nodes collaborate in a peer-to-peer fashion and collectively schedule submitted jobs. This inherently decentralized design eliminates potential performance and reliability bottlenecks in the system, and since it spreads the load associated with job scheduling evenly amongst all nodes in the P2P overlay, it allows the system to scale.

We evaluate $DGS_{ASAP}$ in a simulator using traces from Grid'5000 [3]. We show that our approach—despite its simplicity—allows very efficient job scheduling, achieving a Grid utilization above 90%. Moreover, the maintenance overhead in our model is fixed. The total number of messages transmitted per node depends on the frequency and size of submitted jobs but does not depend on the overall system size.

The rest of this paper is organized as follows. Section 2 describes the related work. Section 3 describes the $DGS_{ASAP}$ model. Section 4 evaluates the performance of $DGS_{ASAP}$. Section 5 discusses future work and concludes.

## II. Related Work

Existing Grid schedulers can be roughly divided into three categories: centralized, hierarchical, and decentralized. Condor [4], originally designed for workstation environments and later extended to Grids, is a widely-known scheduling system that falls into the former category. Each node in Condor has a local scheduler which manages local tasks and records machine idle time, but a centralized *coordinator* polls all the nodes for idle CPU cycles and assigns jobs to available machines. Due to the centralization, this approach does not scale well and is suitable for rather limited numbers of nodes. In a large system, the coordinator becomes a performance bottleneck and a single point of failure.

Consequently, large-scale Grids consisting of multiple sites usually use hierarchical schedulers (also called metaschedulers) such as KOALA [5] and Gridway [6]. In these approaches, each site has its own scheduler that coordinates local resources and accepts jobs from local nodes. When receiving a job, the site scheduler acts as a metascheduler—it polls the schedulers from other sites in the Grid for available resources and requests job execution. Thus, the metascheduler suffers a similar drawback as the centralized coordinator in Condor: when a site scheduler fails, none of its nodes can submit jobs nor can be allocated for job execution. Hierarchical schedulers

also raise scalability and performance concerns when the grid sites grow in size and number.

In order to achieve better fault-tolerance and scalability, the research community has proposed decentralized Grid schedulers. Zorilla [7] is the most representative example of such a scheduler based on a P2P overlay. In Zorilla, a node can directly submit jobs to another node in the overlay without using any designated coordinators. In order to discover resources available in the Grid, a node floods its proximity in the overlay with search messages. Though robust, this approach has two major drawbacks. First, nodes do not maintain schedules and allocate resources opportunistically. If a node is not able to discover resources available immediately, it delays job execution and re-runs the search algorithm. Second, the flooding algorithm generates a high number of messages, growing exponentially with the search horizon, which makes Zorilla rather expensive compared to traditional schedulers.

Fiscato et al. [8] advocate an approach where a combination of multiple specialized "helper" overlays are used to efficiently search for nodes in the Grid. Each of such helper overlays can be optimized for a different search criterion. Specifically, they use an overlay based on CYCLON [9] for random overlay exploration and a proximity-based overlay to discover nodes located as-close-as-possible (ACAP) to each other. Interestingly, they suggest that it is possible to support ASAP scheduling by clustering nodes whose first available time slots in the schedules largely overlap, but they do not pursue this suggestion. This approach is similar to ours, but there are two major differences. First, ACAP scheduling is suitable for network-bound and I/O-bound jobs, while we are focusing on ASAP scheduling for compute-intensive jobs. Second, our model is simpler and cheaper, since we use only one inexpensive P2P overlay.

## III. SYSTEM MODEL

In $DGS_{ASAP}$, all nodes participate in an unstructured P2P overlay, which enables any node to issue a job request to the Grid. Each node maintains its own local job schedule, which describes the jobs that this node is going to execute. Local schedules are composed of time slots, i.e., equal-length time intervals, which can either be free or assigned to a particular job. We assume that a node can execute only one job at a time.

We are focusing on compute-intensive jobs, where the location of resource nodes is of low importance, but the challenge is to execute the jobs as soon as possible in order to minimize their completion time and achieve high Grid utilization. We assume in this preliminary study that jobs do not distinguish between node types. This simplification is partly justified by traces from existing large-scale Grids, which indicate that users rarely require specific properties at resource nodes and typically issue exclusive reservation requests for a certain number of nodes for a fixed amount of time [11].

In our model, a job request $r$ is a triplet $\langle t, n, s \rangle$, where $t$ stands for the time of submission, $n$ is the number of requested nodes, and $s$ is the requested number of time slots per node. For each submitted job request $r$, $DGS_{ASAP}$ produces a job allocation which is a pair $\langle t', N \rangle$ such that $t'$ is the job start time and $N$ is the set of nodes allocated for the job execution. We require that all instances of job $r$ must start concurrently at all allocated nodes and run uninterruptedly for $s$ time slots, and thus all nodes in $N$ must have free schedules from time slot $t'$ up to $t' + s$. The difference between the job submission time $t$ and the job execution time $t'$ is called *waiting time*. The goal of $DGS_{ASAP}$ is to find for each submitted job request an allocation that minimizes the waiting time. In other words, it attempts to execute jobs as soon as possible.

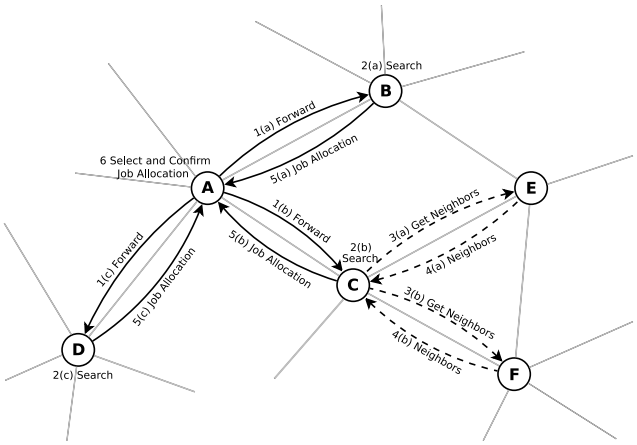



Fig. 1: $DGS_{ASAP}$ overview

Figure 1 shows an overview of $DGS_{ASAP}$. We assume that each node has a knowledge of the job schedules of its immediate neighbors in the P2P overlay. The algorithm that allows nodes to maintain such a view is described later in section III-C. In order to schedule a job request, node A forwards the request to three of its neighbors: B, C, and D. The receiving nodes search in parallel through their schedules and the schedules of their neighbors to construct appropriate job allocations (steps 2(a), 2(b), and 2(c)). If any of the nodes does not have enough neighbors to accommodate the request, such as node C, it obtains the schedules of its neighbors' neighbors (steps 3 and 4). The design of the search algorithm is described in more detail later in section III-B. Eventually, all nodes B, C, and D return their candidate allocations to node A, which selects the one with the lowest waiting time and confirms it.

### A. Peer-to-Peer Overlay

We chose an unstructured P2P overlay as a communication substrate in our system since these overlays are known to be very highly resilient to node failures and network partitions and are rather inexpensive to maintain [10]. They also do not require any centralized management. Specifically, we use an algorithm similar to CYCLON [9] to generate and maintain a P2P overlay. In this algorithm, each node stores a small cache containing addresses of a few other randomly selected nodes in the system. Nodes periodically exchange their caches to randomly shuffle the connections in the overlay and to discard nodes that have left the system or failed. In order to join the system, a node contacts any other node in the overlay and

INPUT: job $r = \langle t, n, s \rangle$ submitted at time $t$, requesting for $s$ time slots at $n$ nodes
OUTPUT: job allocation $\langle t', N \rangle$, where $t'$ is the job start time ($t' \geq t$) and $N$ is the set of allocated nodes

1) Select $fwd$ random neighbors and forward request $r$ to them (each of these neighbors runs the Search Algorithm shown in Figure 3)
2) Receive $fwd$ job allocations
3) Select job allocation with the lowest waiting time and confirm it (each node involved updates its schedule)

Fig. 2: Scheduling a job request.

obtains a few initial addresses to seed the cache. Through cache exchanges, the joining node discovers other nodes in the system. If a node fails, it stops participating in the cache exchanges and is gradually evicted from the nodes' caches.

### *B. Scheduling a Job Request*

Figure 2 shows the scheduling algorithm executed when a job request $r$ is submitted to a node. The scheduling node forwards the request to $fwd$ randomly selected neighbors, where $fwd$ is a system parameter that trade-offs scheduling accuracy (in terms of waiting time) for the scheduling cost (in terms of required computation and network communication). Typically, we used an $fwd$ value of 5 in our experiments. The selected neighbors perform in parallel the Search Algorithm described in Figure 3 which produces candidate allocations for job $r$. The scheduling node receives these candidate allocations and selects the one that has the lowest waiting time. Next, it notifies all the allocated nodes and provides them with the data needed to run the job. The allocated nodes commit to job execution and update their schedules. In case their schedules have been updated by a concurrent job request, they return an exception to the scheduling node, which chooses an alternative job allocation. Similarly, the scheduling node drops an allocation if one of the involved nodes becomes unresponsive. For simplicity, these optional steps are not shown in Figure 2.

Figure 3 shows the pseudocode for the Search Algorithm. It takes a job request $\langle t, n, s \rangle$ as its input and generates a corresponding job allocation $\langle t', N \rangle$ as its output. The algorithm consists of three parts. First, the executing node attempts to find $n$ nodes available immediately for $s$ time slot intervals amongst its direct neighbors (steps 1 to 3). Since each node has a cache with its neighbors' schedules, this part of the algorithm can be run entirely locally, i.e., without any network communication. If it succeeds, the job can be executed with a zero waiting time.

Otherwise, the executing node extends the set of candidate nodes it can use to construct the job allocation by iteratively querying its neighbors for the schedules of their neighbors (step 4). This way, the node can collect the schedules from nodes located up to two hops away from it in the P2P overlay. In our experiments, we typically set node degree to 20, and

INPUT: job request $\langle t, n, s \rangle$ submitted at time $t$, requesting for $s$ time slots at $n$ nodes
OUTPUT: job allocation $\langle t', N \rangle$, where $t'$ is the job start time ($t' \geq t$), and $N$ is the set of candidate nodes

1) Create a set of candidates $C$ containing all neighbors
2) Find a subset $C'$ in $C$ containing $n$ nodes that have free schedules from time $t$ to $t + s$
3) IF $C'$ found THEN return $\langle t, C' \rangle$
4) FOR each neighbor $m$ DO
   a) request all neighbors (and their schedules) from $m$ and add them to $C$
   b) Find subset $C'$ in $C$ containing $n$ nodes that have free schedules from time $t$ to $t + s$
   c) IF $C'$ found THEN return $\langle t, C' \rangle$
5) Increment $t$
6) Find a subset $C'$ in $C$ containing $n$ nodes that have free schedules from time $t$ to $t + s$
7) IF $C'$ found THEN return $\langle t, C' \rangle$
8) Go to step 5

Fig. 3: The Search Algorithm that produces candidate job allocations.

hence a node sends and receives up to 20 messages in step 4 of the algorithm and gathers up to 420 node schedules. Amongst these schedules, the algorithm attempts to find a job allocation with a zero waiting time. Note that if the requested number of nodes $n$ is greater than 420, the scheduling node needs to collect more than 420 schedules by further querying candidate nodes and exploring the neighborhood in the overlay beyond the distance of two hops.

Finally, if the node is not able to find an allocation with a zero waiting time, it proceeds to the third part of the algorithm, executed again entirely locally (steps 5 to 8). Here, the node decides to postpone the job execution by incrementing the job execution time $t$ by one time slot and searching again for a possible allocation amongst all known node schedules. If the search is unsuccessful, the algorithm increments the job execution time and searches again. Since the time scale is unbounded, these steps must eventually produce a valid job allocation.

### *C. Messaging Policies*

In the scheduling algorithm, we assume that the nodes have a knowledge of the private schedules of their immediate neighbors in the P2P overlay. This section describes an approach to maintain such a view.

We assume that each node has a cache where it stores the most recently obtained schedule from each of its immediate neighbors. In order to update these caches, neighboring nodes need to occasionally exchange messages. Since this may incur a significant cost, we are interested in a messaging policy that minimizes the number of message exchanges between nodes while keeping their caches up-to-date and allowing correct job scheduling. Finding such a policy is important, since naive

approaches such as flooding in Zorilla can severely reduce the system's efficiency. Specifically, we consider three alternative messaging policies which we call Push, Pull, and Poll:

1) **Push**. A node notifies all of its neighbors each time it updates its schedule, i.e., when it allocates time slots or cancels a job allocation.
2) **Pull**. A node queries its neighbors each time it runs the Search Algorithm described in Figure 3.
3) **Poll**. A node periodically polls its neighbors. This approach implies the risk of using stale neighbor schedules when running the Search Algorithm, which may lead to scheduling failures.

## IV. Evaluation

In this section we evaluate the performance of $DGS_{ASAP}$. We run parametric scheduling experiments in a simulated grid of 5000 nodes interconnected through a random P2P overlay. In order to make our experiments realistic, we feed the simulation with a trace of job requests extracted from a real Grid. Specifically, we use a 2.7-month trace of job requests submitted to Grid'5000 [3], obtained from the Grid Workload Archive [11].

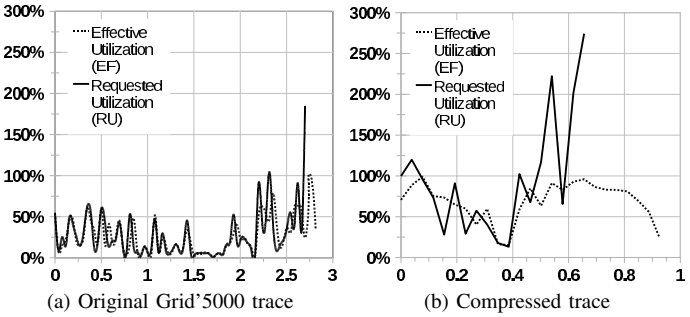
In each experiment, we measure the average job waiting time, the average Grid utilization, and the total messaging cost. Based on these metrics, we evaluate the efficiency of the messaging policies (Push, Pull, and Poll) and we analyze the influence of the P2P overlay on scheduling efficiency. The details of our setup and full results are described in [12].

### A. Scheduling Quality

We evaluate $DGS_{ASAP}$ using three metrics: Waiting Time, Requested Utilization, and Effective Utilization:

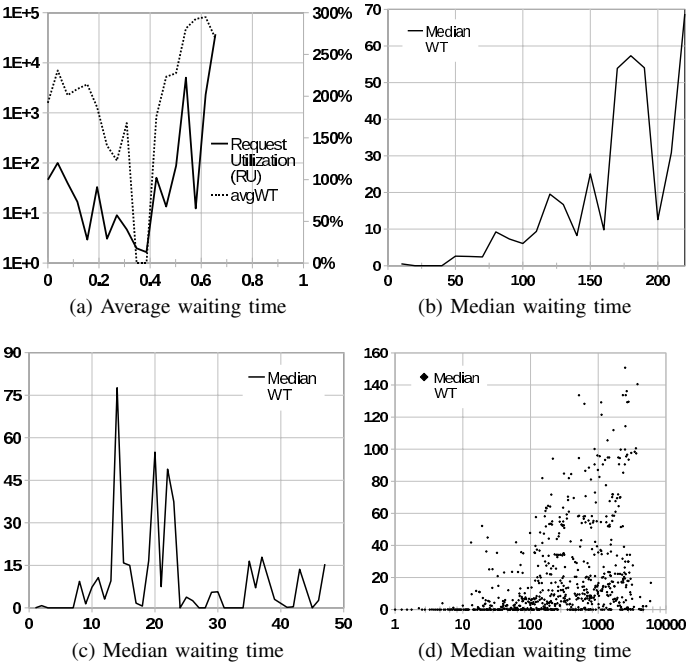
- **Waiting Time (WT)** is defined as the average delay between job submission and job execution start. This is the main metric we use to evaluate $DGS_{ASAP}$.
- **Request Utilization (RU)** is defined as the total number of time slots requested in the Grid during a certain period divided by the total number of time slots available at all nodes in the Grid during that period. This metric measures the load imposed on the Grid.
- **Effective Utilization (EU)** is defined as the total fraction of time slots at all nodes in the Grid allocated for job execution within certain period. This metric measures the average utilization of the Grid resources. We evaluate the scheduling quality of $DGS_{ASAP}$ by comparing EU with RU. Ideally, EU should be equal to RU if RU is below 1 and equal to 1 if RU greater than 1.

Figure 4a shows the requested and effective Grid utilization during the simulated 2.7-month period. Up to time 2.2, the requested utilization is below 60% and RU and EU curves nearly overlap. This suggests $DGS_{ASAP}$ manages to schedule all requests without significant delay. However, at the end of the trace, the requested utilization grows above 100% and nodes have to postpone the execution of some jobs since they cannot satisfy immediately all the requests. The average request utilization in this trace is equal to 28%.




(a) Original Grid'5000 trace (b) Compressed trace

Fig. 4: Request utilization and effective utilization in the Grid. Horizontal axes indicate the time elapsed in months and vertical axes show utilization in percentages.



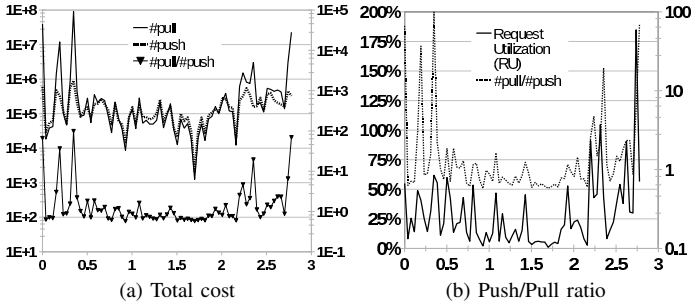

(a) Average waiting time (b) Median waiting time

(c) Median waiting time (d) Median waiting time

Fig. 5: Average and median job waiting time in the 95%-utilization trace as a function of: job submission time (a), number of requested nodes (b), job duration (c), and total number of requested time slots (d).

In order to investigate the behavior of $DGS_{ASAP}$ at higher load, we compress the original Grid'5000 trace by scaling all job submission times such that the average request utilization grows to 95%. As a result, the 2.7-month trace shrinks to about 0.78 months. In the later experiments, we used this trace along with the original Grid'5000 trace to evaluate $DGS_{ASAP}$.

Figure 4b shows the performance of $DGS_{ASAP}$ with the 95%-utilization trace. The RU and EU lines do not overlap, since the system is not able to execute all jobs submitted during the peaks of RU and needs to postpone jobs. At the end of the trace, when request utilization rises above 100%, the effective utilization flattens out and remains almost constant at about 90%, indicating nearly optimum Grid usage. The effective utilization remains strictly positive for several days after the last job submission (i.e., after time 0.78), since the




(a) Total cost (b) Push/Pull ratio

Fig. 6: Comparison between the Push and Pull messaging strategies. The horizontal axis represents the elapsed time measured in months. The vertical axes represent the number of schedules exchanged (a left), request utilization (b left), and the ratio between Push and Pull (a and b right).

nodes need to execute all the postponed jobs.

We conclude that $DGS_{ASAP}$ tends to delay current requests when the request utilization exceeds 60%, resulting in an increased effective utilization in future periods. The overbooking of the Grid at the end of the trace ($RU > 100\%$) causes a slowly declining tail in the EU curve. We observe a similar tail—only much shorter—in the initial experiment shown in Figure 4b. Both tails are the result of postponed requests that could not be scheduled at submission time. During these overbooked periods, the effective Grid utilization exceeds 90%, indicating high scheduling efficiency of $DGS_{ASAP}$.

Figure 5a shows the average waiting time in the 95%-utilization trace. Clearly, the waiting time increases when the requested utilization is high and decreases when the requested utilization is low. This confirms our previous observations that nodes postpone job execution when the Grid is highly loaded.

In order to get more insight into the causes of job execution delay, we analyze the impact of three job properties on the median waiting time: the number of requested nodes, the requested job duration, and the requested number of time slots (i.e., the number of requested nodes times job duration). We use the median waiting time instead of the mean waiting time since the median is not so strongly affected by outliers in our data set as the mean, which allows us to obtain clearer results.

Figures 5b, 5c, and 5d show the results of the measurements. First, the results indicate a strong correlation between the waiting time and the number of requested nodes. Clearly, the waiting time tends to grow for jobs requesting more nodes. Second, there is little or no correlation between the waiting time and the job duration. Finally, the waiting time is generally higher for jobs requesting more time slots from the Grid. Interestingly, $DGS_{ASAP}$ manages to schedule some of the large jobs that request a high number of nodes for a long period of time without any delay.

### B. Message Cost

In order to estimate the message cost of $DGS_{ASAP}$, we count the total number of schedules exchanged between the nodes in the Grid. We do not study possible optimizations for the proposed messaging protocols, such as message batching and broadcasting, although we do not advocate against them. In the Pull messaging policy, a node sends a request message to a neighbor in order to obtain its schedule. Since these request messages can be very small, we do not include them in the total message count. If these messages are added, the message cost of the Pull strategy is simply doubled.

| Policy | Message Cost | Failure Rate |
|---|---|---|
| Push | 152,277 | 0% |
| Pull | 2,187,928 | 0% |
| Poll | 72,000,000 | 0.21% |

TABLE I: Messaging policy comparison.

The number of messages generated by the Push strategy depends on the request utilization (and more precisely on the number of allocated nodes), since a node is required to send a message each time it accepts a job. Moreover, the cost of the Push strategy is proportional to the node degree (number of neighbors), since a node needs to notify all of its neighbors when it changes its schedule.

The number of messages generated with by the Pull policy is generally higher compared to Push, since the Pull strategy requires a node to send a message each time it *attempts* to schedule a job by running the Search Algorithm. Thus, at periods of high load, a node may request many schedules from its neighbors to make a valid job allocation, while only some of these schedules will eventually be changed.

Figure 6 shows the costs for the Push and Pull messaging strategies measured in our experiments. The results confirm our expectations. The ratio between the Push and Pull cost is close to one when the request utilization is low and grows significantly above one (even two orders of magnitude) when the request utilization is high. On average, the cost for Push is over 10 times lower compared to Pull.

Unlike Push and Pull, the total number of messages generated by the Poll strategy does not depend on the request utilization and is known a priori. The cost of Poll is proportional to the number of neighbors per node and inversely proportional to the polling period. Moreover, the Poll strategy implies a risk of job scheduling failures, since nodes may use stale neighbor schedules and may produce incorrect job allocations.

The three messaging policies are summarized in Table I. For the Poll strategy, we use an exchange period of 120 seconds. The Poll strategy suffers a non-negligible job scheduling failure rate, while it incurs an order of magnitude higher cost compared to Push and Poll. The failure rate can be reduced by shortening the polling period, but this would require generating even more messages. Push and Pull do not cause scheduling failures, and as previously mentioned, Push outperforms Pull. Hence, we conclude that Push is the most efficient messaging strategy out of the considered three.

### C. Influence of the Node Degree

In the following experiment, we evaluate the impact of the node degree (i.e., the number of neighbors) in the P2P overlay on the performance of $DGS_{ASAP}$. We run our system four

| Node Degree | Failure Rate | WT | Push Cost | Pull Cost |
|---|---|---|---|---|
| 10 | 1.5% | 247 | 75,592 | 1,108,196 |
| 20 | 0% | 394 | 192,713 | 5,421,255 |
| 30 | 0% | 467 | 282,813 | 11,632,289 |
| 40 | 0% | 452 | 346,302 | 13,097,892 |

TABLE II: Impact of the node degree on the job failure rate, average job waiting time, and messaging costs.

times with node degrees set to 10, 20, 30, and 40. Table II shows the results of this experiment.

For a node degree of 10, some job requests fail because the Search Algorithm is not able to discover enough nodes to construct a valid job allocation—the algorithm can find up to 110 nodes in a 2-hop neighborhood. However, we did not experience any scheduling failures with a node degree of 20 or higher (for the Push and Pull strategies), since no job in the Grid'5000 trace requested for more than 420 nodes.

The lowest waiting time is obtained for a node degree of 10, since in this setup the largest and most difficult to schedule job requests fail and thus do not contribute to the average waiting time. As shown in Figure 5b, such jobs typically suffer large delays. For node degrees above 10, there is no significant difference in the overall waiting time.

Finally, the experiment shows that the message overhead for the Push and Pull strategies grows nearly linearly with the node degree, as expected. As for the Poll policy, the number of exchanged messages is strictly proportional to the node degree by definition and hence is not shown in the table. We conclude that a node degree of 20 yields the best performance for $DGS_{ASAP}$, since it allows nodes to schedule all jobs requests found in the trace and requires a relatively low communication overhead. Consequently, we used a node degree of 20 in all other experiments reported in this paper.

### D. Influence of the P2P Overlay

In order to understand the behavior of our protocol further, we investigate the impact of the P2P overlay on the scheduling performance. We perform two experiments. In one experiment, nodes periodically exchange neighbors, as in the original Cyclon protocol, so that the overlay topology continuously changes. In the other experiment, nodes do not shuffle their connections and the P2P topology is entirely static.

Table III summarizes the results of these two experiments. Interestingly, the waiting time is lower when the P2P overlay is static. We believe, this behavior is caused by a convergence of nodes' schedules. In the static overlay, neighboring nodes often execute jobs together and are thus likely to have similar schedules. Due to this similarity, it is easier for them to find overlapping free time slots, required for job execution. In the dynamic overlay, nodes continuously change neighbors and do not have homogeneous schedules, which impedes job scheduling and forces the nodes to postpone job execution more often.

| Trace | Dynamic | Static | Difference (min) | Difference (%) |
|---|---|---|---|---|
| Grid'5000 | 936 | 848 | 1.47 | 10.38% |
| Compressed | 16,858 | 14,913 | 32.42 | 13.04% |

TABLE III: Waiting times in dynamic and static overlays.

## V. Conclusions and Future Work

This paper studies the feasibility of decentralized and scalable ASAP job scheduling in the Grid. It describes $DGS_{ASAP}$: a model that unlike other Grid schedulers does not have any centralized components and thus avoids inherent performance bottlenecks and single points of failure. It is based on a P2P overlay and a collaborative job scheduling algorithm where loosely-coupled nodes perform only local actions. Since the load associated with scheduling is spread amongst all the nodes in the overlay and does not depend on the overlay size, the system is scalable by design.

Our initial experiments based on a 3-month long trace of Grid'5000 job requests suggest that $DGS_{ASAP}$ can schedule jobs efficiently, achieving a high (above 90%) Grid utilization. In the future, we would like to implement $DGS_{ASAP}$ for a real Grid to further validate its design and to explore potential implementation trade-offs. We would also like to add support for job scheduling based on more criteria than ASAP, while still preserving the decentralized nature of our system.

## References


[1] Lamanna, M.: The LHC computing grid project at CERN. Nuclear Instruments and Methods in Physics Research. 534(1-2) Elsevier (2004)

[2] XtreemOS: a Linux-based Operating System to Support Virtual Organizations for Next Generation Grids, http://www.xtreemos.eu/

[3] Bolze, R., Cappello, F., Caron, E., Daydé, M., Desprez, F., Jeannot, E., Jégou, Y., Lanteri, S., Leduc, J., Melab, M., Mornet, G., Namyst, R., Primet, P., Quetier, B., Richard, O., Talbi, E-G., Touche, I.: Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed. International Journal of High Performance Computing Applications 20(4) Sage Publications (2006) 481-494

[4] Litzkow, M. J., Livny, M., Mutka, M.W.: Condor - A Hunter of Idle Workstations. In: Proc. of the 8th Conference on Distributed Computing Systems. IEEE (1988) 104-111

[5] Mohamed, H., Epema, D.: KOALA: A Co-Allocating Grid Scheduler. Concurrency and Computation: Practice and Experience 20(16) Wiley (2008) 1851-1876

[6] Huedo, E., Montero, R.S., Llorente, I.: The Gridway Framework For Adaptive Scheduling And Execution On Grids. Scalable Computing: Practice And Experience 6(3) SWPS (2005) 1-8

[7] Drost, N., van Nieuwpoort,R. V., and Bal, H.: Simple locality-aware co-allocation in peer-to-peer supercomputing. In: Proc. of the 6th Symposium on Cluster Computing and the Grid. IEEE (2006) 8-14

[8] Fiscato, M. Costa, P. Pierre, G.: On the feasibility of Decentralized Grid Scheduling. In: Proc. of the 2nd Conference on Self-Adaptive and Self-Organizing Systems Workshops. IEEE (2008) 225-229

[9] Voulgaris, S., Gavidia, D., van Steen, M.: Cyclon: Inexpensive Membership Management for Unstructured P2P Overlays. Journal of Network and Systems Management 13(2) Springer (2005) 197-217

[10] Jelacity, M., Voulgaris, S., Guerrauoui, R. Kermarrec, A-M., Van Steen, M.: Gossip-based Peer Sampling. Transactions on Computer Systems 25(3) ACM (2007)

[11] Iosup, A., Li, H., Jan, M., Anoep, S., Dumitrescu, C., Wolters, L., Epema, D.: The Grid Workloads Archive. Future Generation Computer Systems 24(7) Elsevier (2008) 672-686

[12] Vasilakos, X.: $DGS_{ASAP}$: A Decentralized Grid Scheduler for As-Soon-As-Possible Scheduling. Master Thesis, Vrije Universiteit (2009)