Large Lab Exercise A

IN4391 Distributed Computing Systems

Authors:
Yorick Holkamp <yorick@holkamp.net>
Maikel Krause <maikelkrause@gmail.com>

Support:
Dr. ir. Alexandru Iosup
ir. Yong Guo

Abstract
WantDS BV is interested in the research of multi-cluster systems and has decided to invest in the design and implementation of a simulator: the virtual grid system simulator (VGS). We have designed a distributed version of a VGS, which includes a system of multiple schedulers with one elected master node to perform matchmaking. All system state is replicated over all nodes, and the system is fault-tolerant in the face of random node failures. Experimental results show decent scalability results, with minimal additional scheduling overhead in a 4x scale-up.

Table of Contents

1. Introduction

WantDS BV is interested in the research of multi-cluster systems, and has decided to invest in the design and implementation of a simulator of multi-cluster systems: the virtual grid system simulator (VGS). The multi-cluster system that needs to be simulated is a distributed, replicated, and fault-tolerant system. Furthermore, the simulation itself will be distributed on a physical distributed system for performance reasons. In this report, we will design, develop, experiment with realistic scenarios, and report on the feasibility of such a distributed VGS.

There is a non-distributed version of a VGS available. We've decided to use parts of this design as a base. Most of the work is in extending this design to allow scheduling without a centralized grid scheduler.

The system we will implement consists of one Java process per scheduler or resource manager, each simulating a node in our system. Nodes communicate using a message passing system implemented with Java RMI. Nodes may fail at any time, and its work will be recovered by the remaining nodes.

In this report, we will start by expanding on the background and requirements of the desired system in chapter 2, followed by a detailed system design in chapter 3. In chapter 4, we perform a number of experiments to test the system, followed by a discussion and the final conclusion.

2. Background on Application

The VGS application should simulate a grid consisting of multiple clusters and a grid scheduler (GS). Each cluster has a resource manager (RM) and multiple processors. The grid scheduler is distributed, i.e. it consists of multiple server nodes (GS nodes). For performance, the simulation performed by the VGS is itself also distributed on a physical distributed system.

All events (including jobs and system messages) are logged, and replicated across all GS nodes. The system becomes eventually consistent as the logs are propagated to all the nodes. The application should ensure sequential consistency, that is, there needs to be a mechanism to log events in a consistent order.

The application should exhibit a high level of fault tolerance. Any GS node or RM may crash or restart at any time without losing any jobs.

The application should be decently scalable. It should be able to function with the desired properties in a scenario with 20 clusters with at least 1,000 nodes total, 5 GS nodes, and a workload of at least 10,000 jobs. The ratio of the numbers of jobs arriving at the most and least loaded cluster should be at least 5.

Finally, the VGS application should exhibit decent performance even with a high workload. The grid scheduling should not become a major bottleneck, which would defeat the purpose of load balancing jobs in the first place.

3. System Design

System overview

Scheduling architecture

To keep the system simple, but still performant, scalable, and with decent load balancing, we've decided to go with a hybrid decentralized/matchmaking architecture. All GS nodes are in principle equal, but at any time one GS node is elected to be the "master" node. The master performs a number of centralized tasks such as event sequencing and diffusion, and matchmaking GS nodes to RMs.

To prevent the master node from becoming a single point of failure, we need a special protocol to let the system recover from master node crashes. When an RM tries to connect to the master node, but fails (due to a crash or restart), it will temporarily buffer all jobs and events. The crash will be quickly picked up by other GS nodes, who will elect a new master node. Once elected, all RMs are informed of the new master node and activity can continue.

Communication model

For communication, we rely on a Java RMI middleware layer. This saves us from having to implement our own lower level message passing middleware. We believe relying on an existing middleware layer provides us with enough functionality without giving too much overhead.

When any event occurs, a message regarding this event is sent to the master node, which then broadcasts this message along with a unique sequence number to every GS node. Through this method we can guarantee the sequential ordering of events in our system.

Protocols

Bootstrap

When starting the system, the GS nodes get initialized first. One GS node will arbitrarily be named the master. Next, each RM can join the system in turn by sending a join message to any

GS node, who will respond with the identity of the current master. Upon joining, the master matches each RM to one of the GS nodes by choosing the GS node with the lowest load. This load is calculated by looking at the number of jobs in the job queues of the currently matched RMs. The master programmatically gets a higher load to compensate for the extra work it needs to perform.[1]

Adding a new job

When an RM adds a new job to its job queue (and doesn't offload it), it notifies the master of this event. This information is passed to all other GS nodes to allow the scheduling algorithm to work with minimal load polling.

Offloading a job

When an RM wants to offload a job, it notifies the master. The master will notify the GS node matched to this RM to perform the actual scheduling.

Joining of an RM

When a new RM wants to join the system, it will broadcast a join request to each GS node.[2] Each GS node will register the new RM in its internal bookkeeping. In addition, the master node will perform the matchmaking procedure described in the bootstrap.

Leaving of an RM

When an RM is detected to have failed, this event is sent to the master node and broadcast to all GS nodes. The RM is then no longer used to assign jobs to anymore and any jobs that were known to be running are then marked as lost and processed as new jobs that need to be load balanced.

Joining of a GS node

When a GS node wants to join, it broadcasts a join request to each other known GS node. Only the master will respond, which will bring the new GS node up to speed by informing the node of all RMs online, as well as the entire system log.

Leaving of a GS node

A GS node may leave at any time (for example due to a crash or restart). The absence of a GS node will be detected by the master node in the form of a time-out on an expected acknowledgement,[3] at which point it will start an election to find a node suitable to pick up the workload of the node that failed.[4]

[1] cf. `GSNode.getLeastLoadedGSNode()`
[2] cf. `RMNode.connectToGridScheduler()`
[3] cf. `GSMessageChannel.deliver()`
[4] cf. `GSNode.fillInForDeadNode()`

Additional features

Advanced fault-tolerance

Due to our approach where the system state is replicated between all GS nodes the system is resilient against multiple simultaneous failures. While the system recovery time may take slightly longer to the additional work required to recover the workload of the failed nodes, the system will recover without loss of data. We will describe these properties further in the next chapter in the paragraph dedicated to the ‘Fault-tolerance’ experiment.

Benchmarking

After we developed our *StatusChecker* tool and executed the Utopia experiment, we discovered what factors do or do not seem to have a significant impact on the system performance. We noticed that the CPU load of the master node appears will be the limiting factor in the system. This load is caused by the amount of messages that have to be broadcasted to the other GS nodes in the system.

Stressing this part of the system can be done by either increasing the amount of GS nodes - as this means a message has to be delivered to n additional recipients - or by increasing the amount of jobs being executed in the system. As we will describe in the ‘Scalability’ experiment outlined in the next chapter, we decided a scenario with 4 times the amount of jobs and clustes would be both sufficiently stressful and realistic for real world applications.

4. Experimental Results

Experimental setup

We use DigitalOcean[5] to run our experiments, which is an IaaS provider much like Amazon EC2 but offering services at a lower price point.

[5] https://www.digitalocean.com

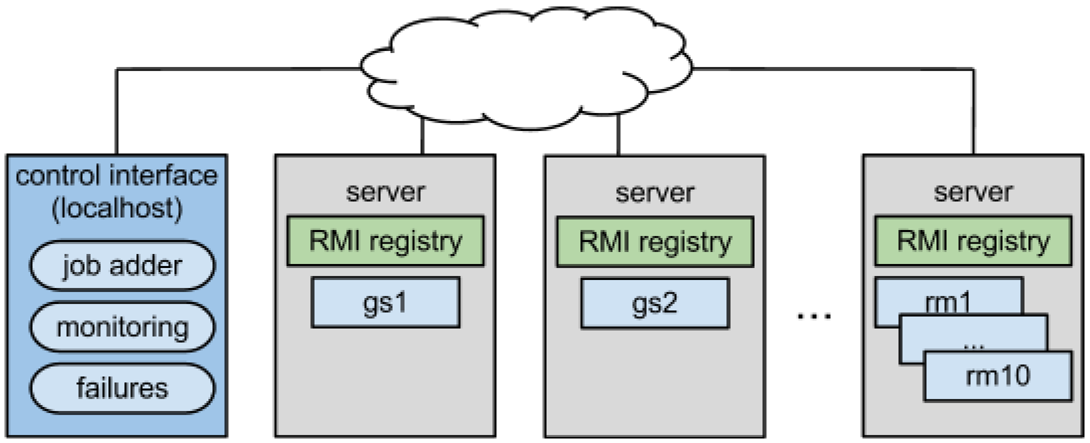
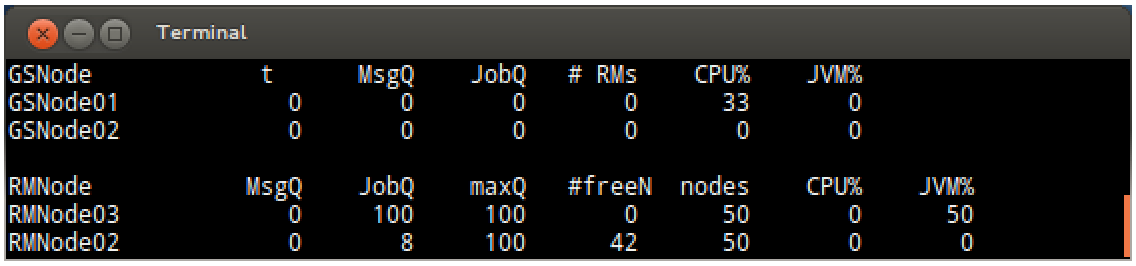


Figure: Typical experimental server setup

Each GS/RM node is a separate Java process, which can run on any of the servers we set up through DigitalOcean. To save costs, and provide more flexibility, we decided to allow multiple RM nodes to run on a single physical server (see figure) but opted to give every GS node its own machine for performance reasons. In practice this meant that up to 10 RM nodes were running on a machine with 2 CPU cores, while the GS nodes ran on their own single core system.

On our local machines we can run several control and diagnostic mechanisms. To start an experiment, we use a batch script[6] to connect to each server over SSH, and start the relevant processes in their own `screen` sessions. The *StatusChecker* tool can be used to monitor the system in real-time, and the *RMIJobAdder* tool allows us to inject new jobs into the system. We can also simulate failures by connecting to one of the `screen` sessions and manually quitting a process.


```
GSNode           t      MsgQ     JobQ    # RMs    CPU%     JVM%
GSNode01         0         0        0        0      33        0
GSNode02         0         0        0        0       0        0

RMNode        MsgQ      JobQ     maxQ   #freeN   nodes     CPU%     JVM%
RMNode03         0       100      100        0      50        0       50
RMNode02         0         8      100       42      50        0        0
```

Figure: RMIJobAdder tool in action, showing the system status during system operation.

GS/RM nodes communicate over Java RMI. In order for this to work, we need to supply each process with the IP addresses of the other servers in the experiment using the `system.properties` configuration file.

Debug logs are printed to `stdout` while the experiment is running (for interactive monitoring), while simultaneously writing to a log file for later analysis. In addition, the complete system log

[6] cf. `experiments/batch.sh`

can be written to file when the experiment has finished.

Experiments

Base configuration

- 20 clusters with 50 nodes each (1,000 nodes total), with a local job queue with room for up to 50 pending jobs.
- 5 GS nodes
- A workload of 10,000 jobs with a random duration of 10-30 seconds.

Experiment 1 – Utopia

The first experiment we executed is the most trivial case. We run the base setup without any simulated failures in order to assess the desired properties of the system under these conditions. The entire workload is added in one large burst. Below we give a summary of the results.

Correctness and consistency

First, we looked at the correctness of the system. Through the system log we know that all 10,000 jobs were correctly completed. We also compared the system message logs between GS nodes, which proved to be identical for all GS nodes, confirming the consistency of the system as the state of any node in the system follows from the messages that have been sent.

Performance

Next, we looked into the performance of the system. An important metric here is the time a job spends in the scheduler, we want this to be as low as possible since we don't want the scheduler to become a bottleneck. The results of these measurements can be seen in the figure below, which shows the time spent in the job scheduler for successive jobs.

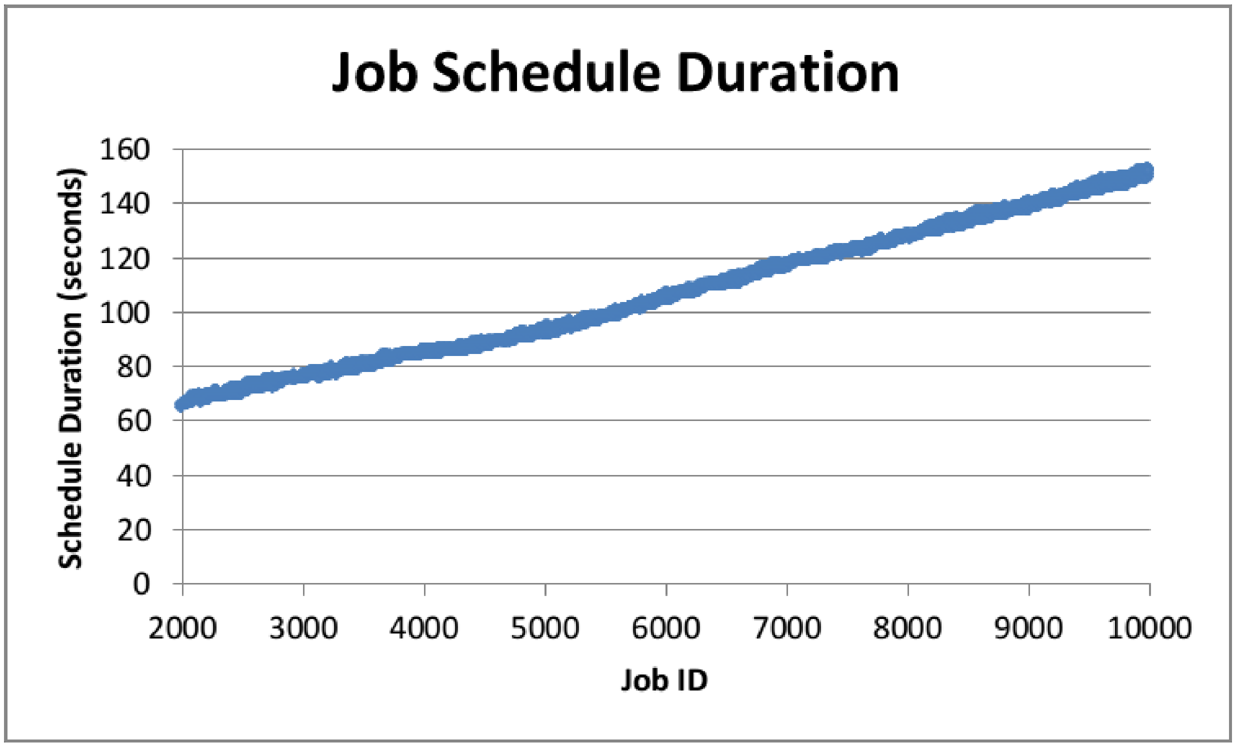


Figure: Time spent in the scheduler by successive jobs

In addition to the time jobs spent in the scheduling process, we also measured the job queue size throughout the system while it was active, as can be seen in the figure below. What we can see here is that in the first 60 seconds of the experiments the jobs were arriving at the various RM nodes, which caused their job queues to rapidly fill to the maximum size. From this point on jobs are being offloaded until about 60s into the experiment. Here we see that as no more jobs are being added to the system, the job queues are shrinking. From around 80s we see that the RM nodes are working through their job queues and receive jobs from the GS nodes to process. What we also see here is that our scheduling algorithm might be too careful not to overload the RM node queues while assigning jobs. In practice this means that the RM nodes appear to have 5-10 nodes idle at the measurement intervals. Through some minor tweaks we could eliminate this overhead and further improve the performance of the system.

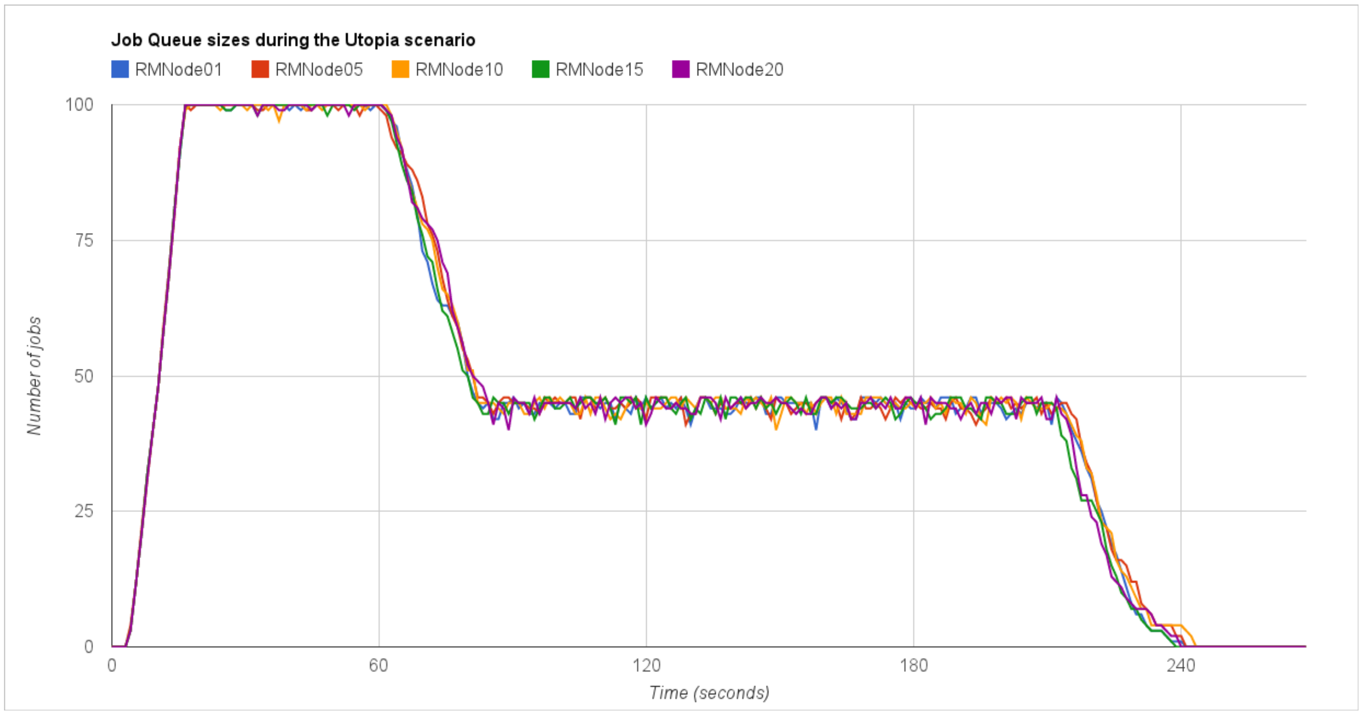


Figure: Job queue sizes on five of the RM nodes during the Utopia run.

Finally we observed that in order to handle the processing of these 10,000 jobs, a total of 25,819 unique messages been sent. This includes 10,000 messages upon completion, 5,775 offload and assign messages and 4,225 regular (non-offloaded) job start messages and 44 additional system messages, used to check whether nodes are still alive.

Scalability

Finally we also measured the CPU usage of the JVM processes in order to get some insight into the scalability of our system. We found that the CPU usage of the master node appears to be the most significant bottleneck of our system; at times the CPU usage of the JVM reaches 100%. This higher load of the master node is caused by the workload of sequencing messages and passing these messages on and this traffic peaks in the first 60 seconds of the experiment as at that time the 10,000 jobs were being added to the system.

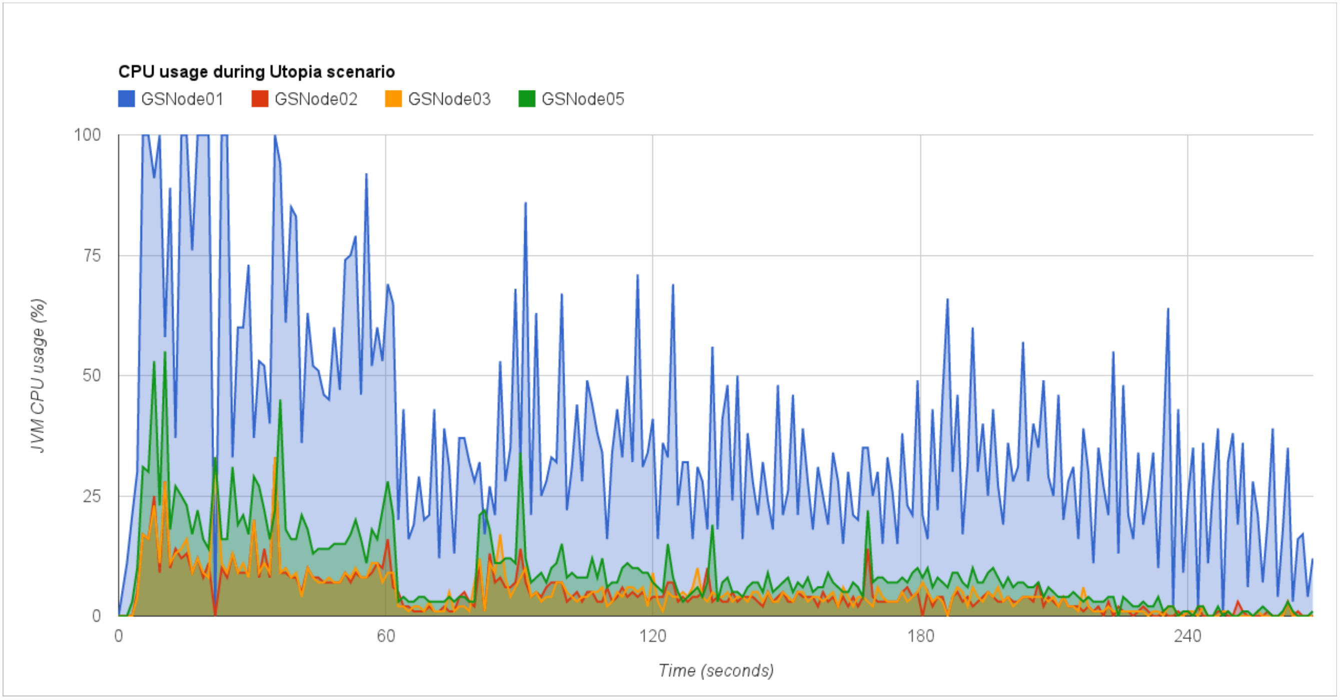


Figure: CPU usage on the GS nodes during the Utopia run. Note that GSNode04 was omitted as this machine was also used to execute the RMIJobAdder and StatusChecker, which skewed the measurements.

The design used in the system makes heavy use of threads for scheduling, handling incoming messages and delivering new messages. As our GS nodes were tested on single core machines, this heavy use of threads might in fact impose a performance penalty in this scenario, as the CPU will spend a lot of time switching contexts. When deployed on a multi-core or hyper-threading machine this penalty will be negated and we believe the performance will scale very well with the amount of additional resources.

Experiment 2 – Fault-tolerance

We extend the previous experiment by inducing some failures. We let the workload propagate through the system so all the relevant queues are non-empty. In this state, we simulate failures by manually interrupting certain processes. We differentiate five kinds of failures:

1. Master GS node failure
2. Regular GS node failure
3. RM node failure

After running this experiment, we analyzed the system logs and discovered that in each case, the jobs were correctly recovered. The failure was discovered and the least loaded GS node took over for the deceased node, including its master duties in case 1, its matched RM nodes and scheduling queue in case 2, and its job queue in case 3. In addition we observed that in the case of simultaneous failures, such as case 1 and 2 or case 1 and 3 combined, the system recovers correctly without loss of data.

Recovery for a master GS node is handled in the order of a few seconds and the recovery of the workload of a regular GS node on the order of a second. For longer-running experiments we expect recovery time may become an issue, as it would take more time the longer the experiment runs due to the necessary synchronization of the system log. For our purposes here this is acceptable, but in real-world scenarios we would want to be more clever in storing and synchronizing the system log in the long term.

Experiment 3 – Scalability

Finally, we want to test the scalability of the system. In a real-world scenario, we expect the number of clusters to grow to potentially very large numbers, while the scheduling infrastructure remains relatively static. Here, we test this by quadrupling the number of clusters to 80 nodes, keeping the same number of grid schedulers. The workload scales accordingly to 40,000 jobs.

In the utopia scenario, it took 236s from adding the first job to the completion of the last job. In the scaled-up scenario, this took 262s. This translates to an additional overhead of 11.1% for a scenario with 4 times the number of clusters, which seems reasonable considering the large increase in communication among the GS nodes.

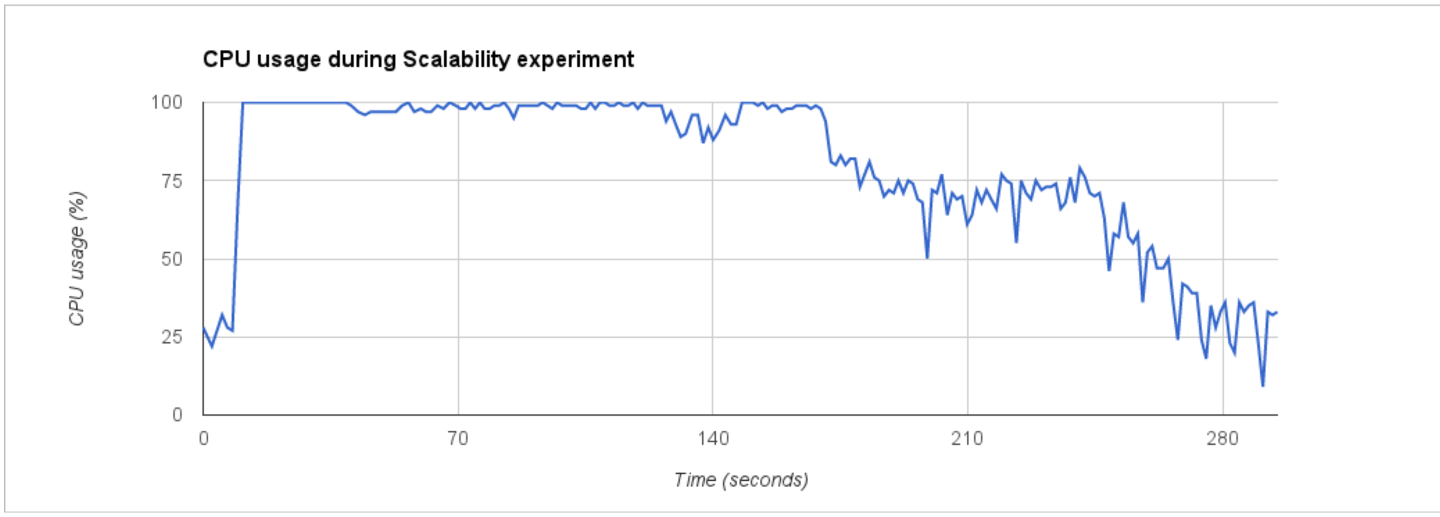


Figure: CPU usage during the Scalability experiment for the machine hosting the master node.

When we look at the resource utilisation during the upscaled scenario we see that the machine hosting the master node has an average CPU usage of around 90%, peaking at 100% during the time the jobs are being added to the system, which means the single CPU core of the machine hosting the master node is a definite bottleneck for our system. On the other hand, the machines holding the other GS nodes show CPU usages in the range of 20-45%, suggesting the resources available pose no limitation on the performance of the regular GS nodes.

Usage metrics and costs of the experiments

In addition to the previously shown CPU usage charts we measured the total CPU usage of the two dual core machines holding the RM nodes, producing the graph below.

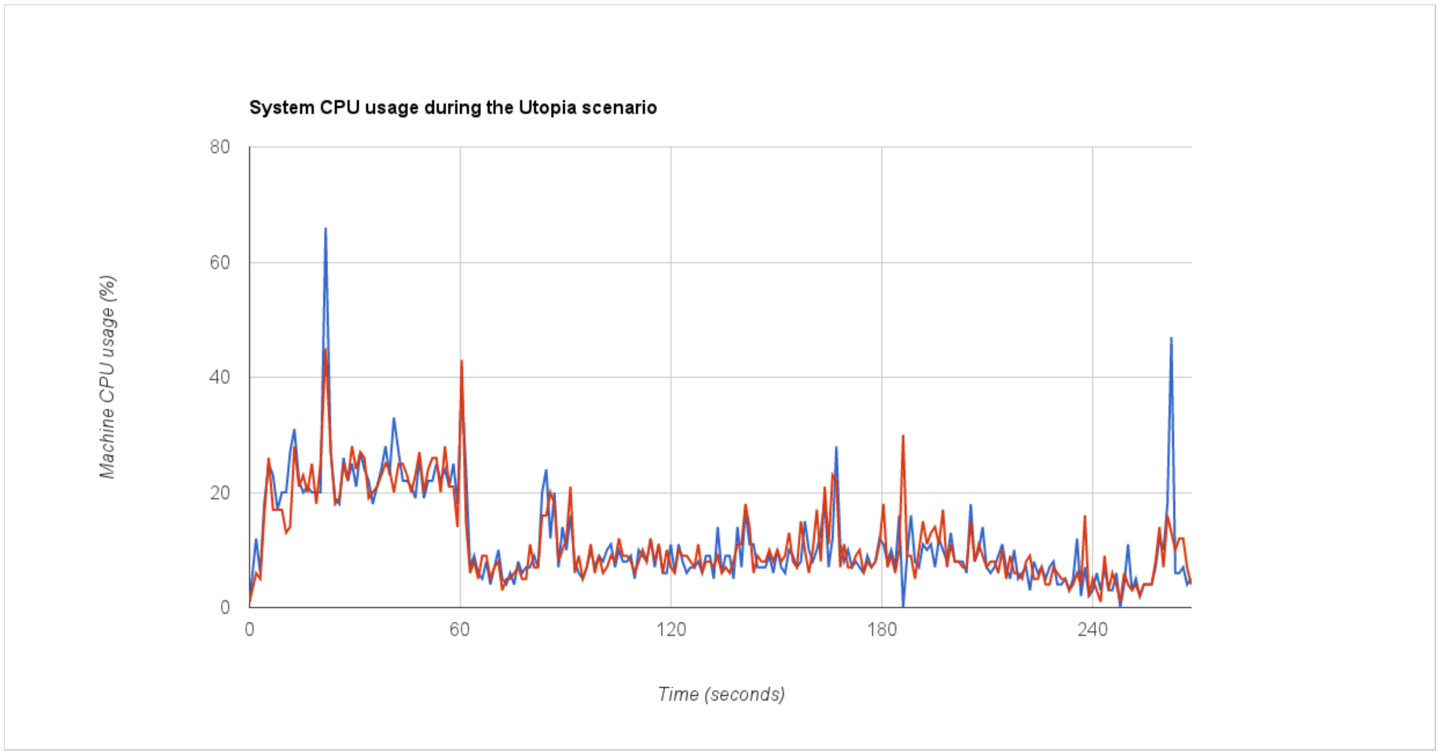


Figure: CPU usage on the two dual core machines hosting 10 RM nodes each.

What we see here is that the CPU usage on the RM node machines is rather moderate throughout the experiment. Except for two of the blue spikes at the start and end of the experiment, the resource utilization is rather low. We can easily see that we would be able to host far more RM nodes on each machine, at least the double amount and perhaps as much as 4 times the current amount, if we were to maximize the utilization of these machines.

Over the course of a few days we used 5 single core and 2 dual core machines to conduct our experiments. In total we used these single core machines for 405 hours, the dual core machines for 114 hours. The total costs add up to $6.39.

Source code release

We released the source code on GitHub, along with this report on GitHub:
https://github.com/yholkamp/dgs2
For instructions on compiling and running the project, see the included README.md.

5. Discussion

We have seen how we can build a distributed virtual grid scheduler and through experiments we have been able to get a good glimpse of the characteristics of such a system during usage.

Compared to the non-distributed variant, a distributed VGS is much more complex. As an indicator, the difference in size of the respective code bases is quite drastic. This complexity also translates to reduced performance, a lot of work goes into just keeping the system consistent through message passing, which is time that is not spent actually performing the scheduling.

Despite these trade-offs, we get a system which is flexible, scalable, fault-tolerant and still shows decent performance. Through the experiments we've seen that our 10,000 job configuration posed no real problem under expected conditions and still performs well in a 4X scale-up scenario while using machines that offer limited resources. At some point, we expect the managerial tasks of the master may become a bottleneck. We believe that by deploying the GS nodes on machines with multiple CPU cores the system will be able to handle workloads of several magnitudes larger size without issue.
Further research should look into the options of using a hierarchical approach, which would reduce the amount of message passing the master has to perform at the cost of additional system complexity, or fully decentralized approach if scalability has the utmost priority.

In a environment with a high frequency of failures, we expect our recovery procedures, especially for master failures, are not entirely optimal. An improvement could be to introduce a secondary master that can take over directly if the primary master fails, which would remove the 1-3 seconds ‘downtime’ of the system while electing a new master and redistributing the workloads.

Another simple optimization would be to tweak the scheduler's criteria for assigning jobs to RMs, which proved too conservative in our utopia experiment. Additionally the system could be improved in terms of performance by implementing a routine to periodically redistribute the matched RMs depending on the system load of each GS node rather than through initial matchmaking.

Finally we would not recommend using Java RMI to develop a distributed virtual grid scheduler without being fully aware of all the drawbacks and limitations it imposes. Limitations we ran into while testing were for instance the fact that the `rmiregistry` only accepts `bind` requests for objects that are on the same machine, which means every physical machine requires its own registry instance and seemingly random behavior when the registry is not started from the working directory or when nodes do not properly `unbind`. An interesting alternative for this approach would be to use one of the many p2p libraries available for Java, which could work well when implementing the system in a fully distributed fashion as mentioned earlier.

6. Conclusion

We designed a distributed VGS application, which is replicated, consistent, and fault-tolerant. Through our experiments, we find that such an approach is feasible with decently high workloads, without sacrificing too much performance, and with decent scalability properties.

Further research may be needed to test the scalability of the system when given extreme workloads. Some simple optimizations have been identified that may improve the system further.

Appendix A: Time sheets

Date	Start	End	Description	Type	Multiplier
2013-02-26	13:00	16:00	Project start	think	2
2013-03-09	11:05	11:40	Checking out the DAS cluster	wasted	1
2013-03-14	16:20	16:50	Checking out the competition for Amazon EC2	wasted	1
2013-03-14	18:45	22:45	Large exercise A design document / report	write	1
2013-03-15	19:30	23:05	Large exercise A design document / report	write	2
2013-03-16	19:00	22:40	Developed a basic building block for the system	dev	1
2013-03-17	11:30	15:00	Rewrote basic building block using Java RMI	dev	1
2013-03-17	15:30	18:00	Design document / report collaboration	write	1
2013-03-17	19:50	22:00	Design document / report collaboration	write	1
2013-03-17	22:00	22:30	Java RMI rewrite	dev	1
2013-03-18	11:45	12:25	Development	dev	1
2013-03-18	19:15	22:15	Development	dev	1
2013-03-19	11:50	16:30	Development, hunting for race conditions	dev	2
2013-03-19	20:00	22:30	Development	dev	2
2013-03-20	11:00	15:30	Development, peer-bughunting	dev	2
2013-03-20	20:30	23:15	Development	dev	2
2013-03-21	20:00	22:30	Development	dev	1
2013-03-22	11:00	18:00	Development	dev	2
2013-03-23	11:00	18:30	Development	dev	2
2013-03-23	19:15	23:00	Development	dev	2
2013-03-24	12:00	14:00	Development	dev	1
2013-03-24	14:00	16:00	Experimenting with the system on localhost	xp	1
2013-03-24	16:00	18:00	Development	dev	1
2013-03-25	11:00	13:30	Development	dev	2
2013-03-25	15:30	17:45	Experimenting with cloud deployment	xp	2
2013-03-25	21:15	22:00	Development of missing pieces	dev	1
2013-03-26	15:00	17:30	Preparing cloud deployment	xp	1
2013-03-26	20:00	23:25	Development of missing RMI functionality	dev	1
2013-03-27	12:30	15:30	Experimenting & demo	xp	2
2013-03-28	10:40	12:30	Development of system monitoring	dev	1
2013-03-28	10:40	12:30	Report	write	1
2013-03-28	12:45	18:00	Report	analysis	1
2013-03-28	12:45	18:00	Experimenting	xp	1
2013-03-30	11:00	12:30	Report	write	1
2013-03-30	15:00	18:00	Report	write	1
2013-03-30	19:20	23:50	Report + Experimenting	write	1
2013-03-30	20:35	23:50	Report + Experimenting	write	1

Total: 156:40 hours

Type	Amount
analysis	5:15:00
dev	94:10:00
think	6:00:00
wasted	1:05:00
write	29:55:00
xp	20:15:00

Table: Total amount of hours per type of task. Note that the categorization is an approximation as working on a system like this means switching between the various types of tasks at rapid pace.