Distributed Systems
The Virtual Grid System: Distributed Simulation of Distributed Systems

Teaching Staff: dr. Alexandru Uta
VU University Amsterdam
a.uta@vu.nl

Teaching Staff: prof. dr. ir. Alexandru Iosup
VU University Amsterdam
a.iosup@vu.nl

Jayme Bird
VU University Amsterdam
j.bird@student.vu.nl
2603893

Jorik van Kemenade
VU University Amsterdam
j.van.kemenade@student.vu.nl
2607628

Christianne Kappert
VU University Amsterdam
c.j.kappert@student.vu.nl
2517441

Pawel Ulita
VU University Amsterdam
p.k.ulita@student.vu.nl
2604357

Tim Visser
VU University Amsterdam
t.c.visser@student.vu.nl
2603761

Stephen Swatman
VU University Amsterdam
s.n.swatman@student.vu.nl
2606765

December 15, 2017

Abstract

WantsDS BV is interested in the research and implementation of a multi-cluster system, the virtual distributed grid scheduler or distributed VGS. Within this paper we explain the development of the Distributed version of the virtual grid scheduler system designed to schedule jobs with very high levels of scalability while remaining consistent, fault tolerant and maintaining high repeatability. This is achieved through the use of multiple grid schedulers and resource managers, a scalable communication system through zeroMQ message passing and vector clocks for maintaining consistency.

1 Introduction

Industrial and scientific demand continues to grow for massive, large scale distributed systems. Therefore, it is becoming increasingly necessary to research, test and simulate distributed systems using realistic workloads to ensure scalability, fault tolerance, consistency and performance before building and testing the system within a real life production environment. By understanding how a system performs under a simulated environment, hard learned lessons can be achieved before a system is customer facing.

This report details the design, development and testing process for a distributed virtual grid scheduler, which assigns jobs to different nodes on a multi-node cluster computer. A non-distributed gridscheduler is used as a reference implementation [1]. Within the non-distributed single grid scheduler, coordination of all jobs are handled by a single grid scheduler node. This creates a single point of failure as well as limitations regarding scalability, fault tolerance, consistency and performance.

In order to address these issues, we propose a distributed multi-node grid-scheduler where multiple gridschedulers work together to provide scalable load balanced scheduling across clusters, as well as advanced fault tolerance to deal with failures. For scalabilty for node communication we propose using ZeroMQ message passing and for consistency within the system we propose utilizing vector clocks.

In this report we will provide detailed requirements and design required for the system, experiments for testing the system, followed by discussions and the conclusions.

2 Background on Application

The virtual grid scheduler (VGS) consists of a single grid scheduler that handles multiple clusters deciding on which cluster a particular job should be executed. Each cluster has one resource manager that is responsible for scheduling the jobs onto nodes within the cluster, and maintaining a job queue for the cluster. Each cluster has one resource manager (RM), and each RM belongs to only one cluster. Nodes are where jobs are executed, nodes belong to a cluster and each cluster has multiple nodes.

Based on the problem description two main use cases are identified, one initiated by the user, and the other initiated by the grid scheduler. First, the single VGS offers jobs to the resource manager, a job can either be accepted, queued, and executed or offloaded to the grid scheduler if the job queue on the cluster reaches the specified threshold. The job is then offloaded to the grid scheduler (GS), to allocate to a resource manager on a different cluster.The resource manager then allocates the jobs to a particular node within a cluster. Second, an already submitted job can be rescheduled can be rescheduled by the grid scheduler. Whenever the GS sees that a certain cluster is overloaded, it can pull a job from this clusters queue, and schedule it on a different cluster.

Additionally, there are requirements for consistency, scalability, fault-tolerance, and performance. Regarding consistency, the system should report an error message when it cannot be guaranteed that information is up-to-date. The non-functional requirements of the system concern the scalability thereof. The system should be able to handle 5 grid scheduler nodes, 20 clusters, 1000 nodes, and 10,000 jobs in total. Furthermore, the amount of incoming jobs for the most loaded cluster should be 5 times the incoming jobs for the least loaded cluster for a significant period. Concerning fault-tolerance, the system should be able to endure and recover from crashes of the RM and grid scheduler. Also, all system and grid events should be logged on at least two GS nodes in the order they occurs.

3 System Design

The GS node that receives a job from a RM, chooses another cluster, submits the job to the respective RM, and notifies the other GS node. In case the other GS node is not notified about it, it tries to contact the first GS node in order to find out about the situation. If its not possible, it itself submits the job. If the first node manages to be back, it contacts the other one to check the status of this situation. When a GS node tries to move a job from one cluster to another, it contacts the other GS node responsible for rescheduling about it. In case they detect they both want to do it, they wait a random time and try again. When a consensus is reached, they store this information. One tries to store it, and the other one waits for the confirmation. In case no confirmation is received, it tries to poll the first one about the situation. If it fails, it submits the job itself. When the first node manages is back, it contacts the other one to check the status of this situation. An overview architecture of this as designed is shown below in Figure 1.

3.1 Scalability

The reference implementation of a single grid scheduler was only able to handle a few clusters, and had a single point of failure. To improve on the scalability of the design, and manage many clusters at one time, multiple grid schedulers are needed. We implemented a system with five nodes where two are responsible for accepting jobs from resource managers (RMs), two are responsible for balancing the load, and 1 is ready to take over the role of a failed node. Each GS contains its own view of the state of the system with communication between the grid schedulers. This is important for scalability as all nodes need roughly equivalent numbers of jobs in the queue. Communication between nodes is achieved through message passing using ZeroMQ which contain node loads. This ensures scalability in both communication, and queue offloading. Furthermore, the system is very well scalable since the jobs are not real and therefore have low intensity [correct?].

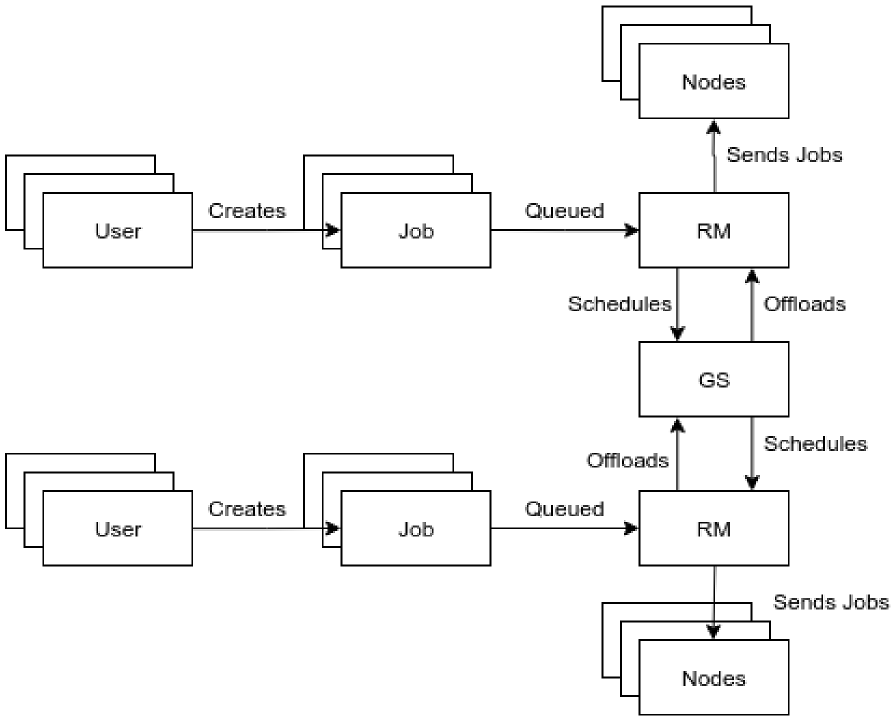

Figure 1: Architectural Overview of the System

3.2 Replication and consistency

The resource manager always replicates jobs. It only returns a 'job accepted' message when it has been replicated to at least one other node. When the RM is unable to return a 'job accepted' message, for example due to too few nodes or a full system, it will return a 'job not accepted' message. This could be fixed by extending the queue length. Besides that, after a number of submitted jobs, the grid scheduler broadcasts this information to the other GS nodes. This should enable the GS nodes to have an overview of the system state that is close to consistent. When a GS node receives a job from the RM, it exchanges this information with the other GS node that is responsible for for accepting jobs from the RMs. Only when it is sure these nodes both have the information, the RM will be notified that the job will be scheduled.

The consistency model used is using logical clocks, and specifically an implementation of vector clocks [2]. This ensures a consistent state between communication and replicated nodes.

3.3 Fault tolerance

The system is fault tolerant making use of the replicated jobs as explained in the subsection above. When a resource manager crashes, all of the nodes attached to it will become unavailable, as specified by WantDS. Alternatively, another cluster could take over. When a grid scheduler crashes, another GS can take over as there are multiple GSs. In a worst-case scenario all GS can fail but one.

Additionally, if there's some communication about to happen, and the destination is down, the sender doesn't crash, but it waits until the recipient is back up,

3.4 Bonus Features

Besides the consistency, scalability and fault tolerance, we completed additional work to improve the features of the system. Specifically, the system can handle multiple failures through advanced fault tolerance. The system has a high level of repeatability and makes sure that the outcomes of a simulation are always the same for the same input.

3.4.1 Advanced Fault Tolerance

Advanced Replication and fault tolerance are crucial aspects of the system, as if a user submits a job and the system acknowledges that the job has been received , a user needs fault tolerance and replication transparency, and does not want to be concerned with the possibility of losing their job, even if multiple failures occur. In the case of multiple failures, the design can withstand an all but one failure scenario as detailed above.

3.4.2 Repeatability

Distributed simulations typically have non-deterministic features that make analysis of logging and runtime observations difficult.

Our system is designed to ensure repeatability of real time distributed simulation executions if there is consistency in the initial execution state, and the input data is the sane. We achieve this through synchronized repeatable message delivery implemented through our consistency model using logical vector clocks timestamps where events containing the same time stamp at any logical process will be processed by that process in the same order from one execution to another, thus ensuring repeatability. [3]

3.4.3 Fault tolerance experiment: Logging

We implemented logging in order to analyse our system and ensure fault tolerance by assessing the behaviour of the system through the system logs, and the performance of consistency in order to recover workloads correctly without loss of data. An example of system logs we collect is shown below

```
LogEntry(timestamp=[8, 15, 85, 105, 97, 0], node='ResourceManager<10001>',
message="Can't accept job ID=1, offloading it to the grid scheduler")
LogEntry(timestamp=[8, 15, 85, 107, 97, 0], node='ResourceManager<10001>',
message='Cannot offload to GS localhost:11000)')
LogEntry(timestamp=[12, 15, 98, 100, 99, 0], node='Node<9990>',
message='Starting job: Job(id=2, duration=datetime.timedelta(0, 5))')
LogEntry(timestamp=[13, 15, 99, 100, 99, 0], node='ResourceManager<10000>',
message='Submitted Job(id=3, duration=datetime.timedelta(0, 5)) to
localhost:9990')
LogEntry(timestamp=[13, 15, 100, 100, 99, 0], node='ResourceManager<10000>',
message='Scheduled job Job(id=4, duration=datetime.timedelta(0, 5))
on the node localhost:9990')
```

```
LogEntry(timestamp=[15, 15, 103, 100, 99, 0], node='ResourceManager<10000>',
message="Couldn't schedule a job, waiting 5s")

User got response JobAcceptedMessage(clock=[0, 0, 54, 60, 80, 0])
User got response JobNotAcceptedMessage(clock=[0, 0, 54, 65, 80, 0])
User got response JobAcceptedMessage(clock=[0, 8, 63, 77, 89, 0])
User got response JobNotAcceptedMessage(clock=[0, 8, 68, 77, 89, 0])
User got response JobAcceptedMessage(clock=[0, 8, 57, 80, 86, 0])
User got response JobAcceptedMessage(clock=[0, 8, 57, 83, 86, 0])|
```

4 Experimental Results

As one of the major requirements for our system is scalability and our system does not schedule jobs when its job queues are full it's important to validate how the implemented job queues work in practice. We set out to investigate whether they perform their task as expected and if they successfully fulfill their function as an intermediate staging area for jobs that are not yet being processed by any node.

4.1 Set Up of Experiment

The experiment we set up in order to test the implementation job queues was designed to meet several requirements. First, we wanted the results to be repeatable and comprehensive in order to be able to provide an accurate and fair overview of the eventual outcome. In order to achieve this, we adjusted our existing user simulation script to our specific needs for this experiment. The simulated user presents a job with a duration of exactly 5 seconds every .5 seconds. This, in our opinion, is a fair simulation of a sudden burst of user activity, but above all is consistently repeatable in order to vary the parameters we are interested in testing the influence of. We use 2 ResourceManager nodes and 1 GridScheduler node for each test. In order to assess the effect of our implemented queue, we vary queue size and number of jobs. We tested queues of size 10, 15, 20 and 30, all of which were tested with 50, 75 and 100 jobs requested as specified above.

5 Results

It is clear from figures 2, 3, 4 and 5 that the queue has the intended result of buffering jobs in times of high burst activity, resulting in reduced rejection rates when load is high, effectively providing a higher job acceptance rate at the cost of higher average job latency. Seemingly the amount of burst jobs that can be handled scales linearly with the size of the job queue as expected. This means it is theoretically feasible to provide a job queue that is high enough to buffer any amount of jobs if this is desired behaviour for the system.

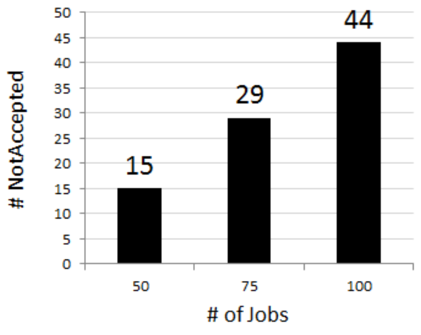

Figure 2: Queue size 10

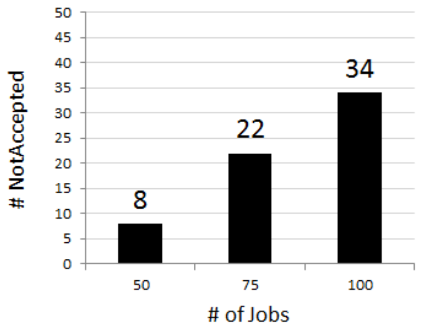

Figure 3: Queue size 15

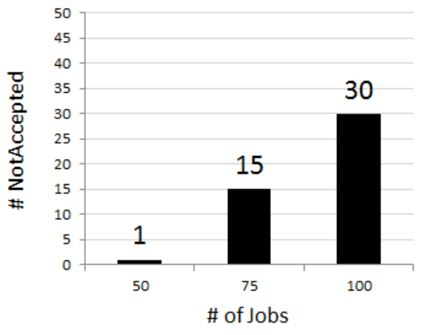

Figure 4: Queue size 20

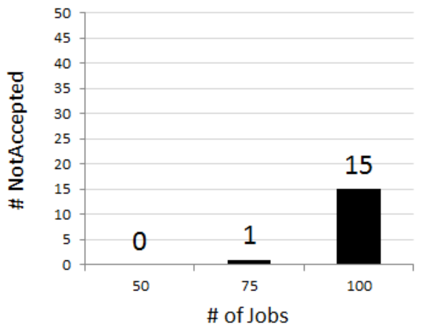

Figure 5: Queue size 30

6 Discussion and Conclusion

6.1 Design Process

The process of designing the Distributed Virtual Grid System was an iterative one. First the task was to implement a non distributed single grid version, and then make design decisions to meet the desired requirements. After each design phase, slight changes were required in the design to ensure requirements were met. Through the process, trade-offs were made in the design when each design decision was made.

6.2 Conclusion

The Distributed Virtual Grid Scheduler is a system that meets all the requirements as a scalable, fault tolerant and consistent scheduler and resource manager for cluster workloads. Furthermore, additional features such as advanced fault tolerance and repeatability exist within the system which are useful for performing simulations of distributed systems for testing and analysis purposes.

The system would benefit from additional testing with real workloads such as the ones collected by the The Grid Workloads Archive [4], as well as on a multi-continent distributed cluster in order to simulate real world scenarios.

References

[1] Ian Foster, Carl Kesselman, and Steven Tuecke. “The anatomy of the grid: Enabling scalable virtual organizations”. In: *The International Journal of High Performance Computing Applications* 15.3 (2001), pp. 200–222.

[2] Mukesh Singhal and Ajay Kshemkalyani. “An efficient implementation of vector clocks”. In: *Information Processing Letters* 43.1 (1992), pp. 47–52.

[3] Richard M Fujimoto. *Parallel and distributed simulation systems.* Vol. 300. Wiley New York, 2000.

[4] Alexandru Iosup et al. “The grid workloads archive”. In: *Future Generation Computer Systems* 24.7 (2008), pp. 672–686.

7 Appendix A: Timesheets

- Total-time: 98h
- Think-time: 20h
- Development-time: 40h
- Experiment-time:5h
- Analysis-time: 8h
- Wasted-time: 5h
- Write-time: 20h

8 Appendix B: Open Source Code and Report

The code and report for the project has been made public and is available at the following URL: `https://github.com/DistributedSystems300/DVGS`