The Virtual Grid System:

Distributed Simulation of Distributed Systems

Team Otso

Iulia Cristina Ban	2604231	iuliacristina.ban@gmail.com
Vasileios - I. Manavis	2593065	v.i.manavis@student.vu.nl
Louise Ivan Valencia Payawal	2591119	livpayawal@gmail.com
Mihail Tomescu	2626024	tomescumihail93@gmail.com
Michel-Daniel Cojocaru	2610955	michel.cojo@gmail.com
Gøran Hovde Strønstad	2604393	goran.hovde@gmail.com

Abstract

This paper introduces a distributed version of the current non-distributed multi-cluster system of WantDS BV. It presents a virtual grid simulator that is composed of several distributed clusters of computer resources which are structured in grids. All of which are managed by different system components such as Resource Manager, Grid Scheduler, and Supervisor node(s) that are basically essential in the context of job allocation, migration, and effective load-balancing across processing nodes. We implemented the system according to the client's requirements. After the implementation, a thorough experimental analysis was performed in order to test the system's behaviour under various conditions. Lastly, by having multiple environments available to perform the experiment simulation, we were able to uncover the factors on how we were able to make the system behave as the client requested.

I. Introduction

Having in mind that resource planning strategies and the adaptation of existing applications to the distributed conditions of multicluster systems are active fields of research, companies may invest in in-house or market-provided products for multi-cluster system simulation.

The client WantDS BV is currently using a non-distributed version of a multi cluster system. They want a coherent high performing system when it comes to running simulations with a large amount of users. WantDS BV is willing to do a trade off between cost and performance to upgrade their system infrastructure. They require the system to be scalable which is usually a constraint factor when using a non-distributed version of a system. As the client infrastructure is shifting towards a distributed version of multi cluster system, they will be able to handle events that are generating large scale transaction, making their system more scalable. This will also boost performance when it comes to response time and concurrency that would lead to a higher system reliability.

Lastly, by creating a distributed version of multi cluster system for WantDS BV, a coherent system that focuses on large scale resource sharing would be generated.

II. Background on Application

The nature of the application is to simulate a distributed multi-cluster system. The constituents of this system are several interconnected grids of computing nodes that together form a distributed virtual grid simulator (interconnected hierarchies of computing nodes). At the top we have a Supervisor node, at the bottom computing nodes and in between, Resource Managers (RMs) and Grid Schedulers (GS). In short, the simulation sets up instances of these nodes, and constantly introduces new jobs at a certain pace. The first point of arrival for any job is the RM job-queue, from which a job is either dispatched to a processing node (if idle node found) or sent to a GS (if idle node *not* found).

In terms of scalability and performance, our application is quite effective in the way we implement load-balancing methods and allow for parameterized input. It is imperative that the operations are synchronized in such a manner that the data remains consistent. The application constantly balances the load across processing nodes, and not just when needed. Developing a distributed system which is one hundred percent fault tolerant is actually extremely challenging. With the introduction of the Supervisor node fault-tolerance is a fact; we have no single point of failure as this node creates (unique) direct communication channels between RM- and GS- nodes. Additionally, GS nodes get replicated in runtime facilitating yet another layer of fault tolerance. In the event of node misbehavior (slow, crashed) the application fluctuates between replica (identical) nodes that are created in the image of the failing node.

III. System Design

Here we describe the VGS architecture by elaborating the components it is comprised of and their interoperation. At the same time, we reason about the design choices and how they serve our requirements. Bonus features are also elaborated, which are Advanced Fault Tolerance and Benchmarking. The latter is described in the Experimental Results section.

Architecture

The architecture of the VGS is mainly comprised of two major components: the *Supervisor* and the *Cluster*. The Supervisor manages the loads of each cluster; it migrates and dispatches jobs from/to the Clusters. The Supervisor, being at the top of the hierarchy, has knowledge of the Grids' condition, this enables it to evenly load the whole system by dispatching jobs from highly loaded Clusters to the least loaded ones. The whole architecture of the VGS is shown in the Appendices.

Clusters

Each cluster is comprised of three basic components: *Resource Manager* and its *Job Queue,* and the *processing nodes* which actually run the jobs. The RM constantly receives jobs which need to be served. Each new job that arrives at a RM is stored at the tail of its Job Queue. The size of the queue is limited, but the queue is persistent. This means that in case of a RM failure, the jobs are not lost - they are just resubmitted.

Resource Manager

Resource Manager is the component of a Cluster that dispatches jobs from its Job Queue to available processing nodes. The policy of serving the jobs of the queue is FIFO. That means that whenever a processing node is made available, RM will dispatch the job at the head of the queue to the node. The job gets served and the result is sent to the sender of the job request (client). In case that the queue gets full, new requests are directed to the linked GS node (see further in the paper).

Supervisor

The Supervisor is the "brain" of the VGS. Its main goal is to even the load of the jobs across the whole Grid. It achieves this by dispatching jobs from high loaded clusters to the least loaded clusters. The Supervisor itself is comprised of several GS Groups, whereas a GS Group is comprised of two linked (among themselves) GS nodes: one Primary and one Replica.

GS Node

GS Node is an abstract idea of a component (statically) linked to various Clusters. The nature of a GS Node is expressed as either Primary or Replica. Every single Primary GS Node is linked to a single specific Replica, and the Replica is linked to that Primary GS Node. Initially, our idea was to link all Clusters to the Supervisor, but in that way we introduced a single point of failure; the Supervisor itself. Instead, each (primary) GS Node is statically linked to a subset of all the Clusters that comprise the Grid.

Each primary GS node is connected to same amount of Clusters (+/- 1). Each GS Node also has a Job Queue from which it dispatches job replicas (simultaneously) to various clusters (at least two). The reason for this replication scheme is explained in the *Fault tolerance and Replication* and *Consistency* section.

Primary

The Primary GS Nodes poll their associated Clusters asking for status on their loads, and from this information make a decision to migrate (move) jobs from RM job-queues (of linked Clusters) to the job queue of the Primary GS Node. A job can also be explicitly migrated from the job queue of a RM in case of job overflow. The function of the Replica is partly elaborated in the *Fault Tolerance and Replication* sections. Furthermore, each primary GS Node has one and only (GS primary Node) neighbor to whom it is linked. That link serves as a channel for migrating jobs from one GS Node to another. All such channels together form a ring which is used to propagate an even load to all GS Nodes. More information on this concept in the *Scalability and Performance* subsection.

System operation

With a general idea of the architecture (Appendix - Figure 1.0 Architecture Overview), we can talk about the system's operation. Resource Managers of the Clusters constantly receive and dispatch jobs from their queues to the available processing nodes. However, the frequency of arriving jobs may be much higher than than the frequency of accomplished jobs. In that case, jobs start accumulating in the RM Job Queue. At the same time, (primary) GS Nodes poll the Resource Managers asking for their load in order to decide if and which jobs to migrate. Only fresh (non-replicated) jobs are migrated. Such jobs are migrated from the RM queues to the GS Node Job Queues. Afterwards, there are two options. The first option is to dispatch them to other (multiple) RMs (that are connected to that GS Node). In that way, a job becomes replicated and it is no longer “fresh”. More about that idea in the next subsection. The second option is to migrate the jobs to another GS Node (neighbor). The subsection of *Scalability* discusses this in more detail.

(Advanced) Fault tolerance and Replication

VGS is also resilient against crashes. Our system allows two type of crashes. Processing node crash and GS Node crash. Both types are handled with unique replication mechanisms. Previously, we talked about replicated and non-replicated jobs. Non-replicated jobs are fresh, newly arrived jobs at the RMs of the Clusters. Replicated jobs are fresh jobs which have been migrated from a RM’s job queue to a GS Node, and from there they have been replicated and dispatched to multiple (at least two) other clusters in order to be executed. Say a processing node fails, then there is (at least) one other cluster that holds the job - and is either waiting to be executed, or is already running. Processing node fault tolerance does not work when all processing nodes where the replicated jobs are running are down. But the chance of simultaneous failure of those specific processing nodes is extremely low, thus it is considered a good way of dealing with that kind of failure. Performance-wise, replicated jobs may add a load on the system but we bet on the faster execution of the jobs. Once a replicated job is executed, a kill signal is sent to all other running instances to release the resources. More info about that in *Consistency* subsection.

GS Node fault tolerance and replication

As stated earlier, each GS Node Group consists of a Primary GS Node and a Replica which are interlinked. The two components share the same Job Queue which does not fail. Each Primary GS Node is (statically) connected to $X = \frac{\#\ Clusters}{\#\ GS\ Groups}$ $(X \in Z)$ Clusters. In case of a (primary) GS Node failure, the linked Replica now becomes the Primary. In order for the swap to be complete, the statically linked Clusters need to be linked to the new Primary (old Replica). This is achieved without any message losses due to the utilization of synchronized sockets which block messages until the swap is complete. The socket acts like a proxy, so not all clusters need to be updated about the new address, but just the socket itself. The Job Queue is persistent and shared between the Primary and the Replica so no worries about the jobs.. Figure 1.1 in Appendix shows the steps of the swapping. If the new primary node fails, then the same swapping takes places. This protocol is based on the role playing of the nodes. GS Node Fault tolerance fails if both the Primary and the Replica fail at the same time. Again, the chance of such a coincidence is really low.

Consistency

Once one uses replication to deal with fault tolerance, it is also highly likely that consistency among the replicated objects is also required. As stated before we have two kinds of replication: Job Replication and GS Node replication. Job replication is applied on a GS Node level. The same Job is dispatched to multiple Clusters which are linked to that GS Node. Note that we do not replicate by creating new deep copy objects. We replicate the job references (pointers). Whichever cluster completes the job first, sends a completion message to the GS Node which sequentially broadcasts that message to all linked Clusters. If the Clusters who receive the message have a replica of the job, then they kill the job and resources are released. The Clusters that do not have a replica of that job, simply ignore it. The overhead of sending the message to all the linked clusters instead of only the ones which have a replica is not so noticeable. The reason is because it makes the consistency protocol really simple (and hence easily maintainable) and it is not required to record each visited cluster. So it's a better approach for consistency and performance. The other example of consistency is expressed with the (primary) GS Nodes and their Replicas. Their function has already been elaborated. However, they are consistent because they share one persistent, non-size limited Job Queue. Another requirement for consistency of this case, is to perform effective swapping. Swapping includes the role playing swap and the Synchronized socket swap. A shared, persistent Job Queue is mainly the key of consistency here because no synchronization is needed due to the idling situation of the Replica.

Scalability and Performance

The scalable components of the system are the Clusters (including the processing nodes) and the GS Groups. Before talking about the behavior of the system as it scales, let's see how the system tries to even the load across the whole grid. As stated earlier in the report, each primary GS Node is linked with a unique neighbor which is a (primary) GS Node of another GS Group. All the links together form a ring. Each GS Node knows the (average) load of the clusters it is connected to. So does its neighbor. Given these loads, we take the average of them and migrate jobs from one GS Node to another so as both of them have the same load. The neighbor will perform the same task with its own neighbor. In that way, the

load of highly loaded GS nodes is progressively migrated to other GS Nodes of the VGS. Figure 1.3 in Appendix depicts how this concept works through various iterations.

IV. Experimental Results

In this section we describe the experimental setup and experiments performed to investigate the behavior of the VGS under certain conditions. Through these experiments we aim to unveil aberrant- or reliable- system behavior.

Experimental Setup

Application development was based on the provided material and took place on a single machine. We adopted two monitoring tools; a Logger and Grid Scheduler Panel. The Logger is obtained from the Apache log4j library and was also used as a debugging method. Experiments also took place on a single machine. Given how we made use of the Grid Scheduler Panel, we could not run the simulation on DAS - as that would rule out the possibility of visually observing the system behavior. Experiments were conducted both programmatically (statically) and on demand.

In Code Experimenting

In the *Simulation* class of the code, one can find the following lines of code. Information on workloads and data procurement can be found in the Appendix.

```
1. idealLoad(jobId++);
2. stressTest(jobId++, ratio: 5);  //The most useful test. It stresses the system with the
   given ration
3. evenLoad(jobId++);          //randomly distributes jobs to cluster (nearly uniform
   distribution)
4. unEvenLoad(jobId++, highLoadTargetCluster, lowLoadTargetCluster,5);
```

On Demand Testing

Apart from those programmatic options, we can also test the VGS on demand by manipulating two major parameters: the job ratio and the duration of the jobs. During the simulation, one can press the up arrow (↑) and increase the ratio on demand and observe how the system reacts. Respectively, by pressing the down arrow (↓), one can decrease the job ratio. Max job ratio is 50. As for the job duration, one can increase the job duration by pressing the right arrow (➡) and decrease it by pressing the left arrow (⬅). Max job duration is 100 seconds.

Experiments

A VGS is a complex infrastructure establishment which offers a lot of areas for experimentation. However, for this assignment we are mostly interested in two major metrics: Job Completion Time (JCT) and Waiting Time (WT). The reason is because these metrics are the only (high level) metrics that the client will be interested in. Actually clients are only interested in the overall JCT, but WT affects JCT the most and therefore it should enjoy special treatment. However, prior and more important to completing a job fast, is to actually complete a job. Therefore, we also perform experiments on how VGS behaves

under crashes and component failures. For each case, we conduct to-the-point experiments that expose the true value of the object under testing. In order to achieve that, we first define the testing object/event and for that we use metrics which reflect that object/event.

To investigate how our system performs, we collect various data on jobs that pass through the system. We log job submission-, wait- and run- time, and from these values calculate the Job Completion Time (JCT) (submission, wait, and run time aggregated). We did this under various workloads (for information on workload see Appendix).

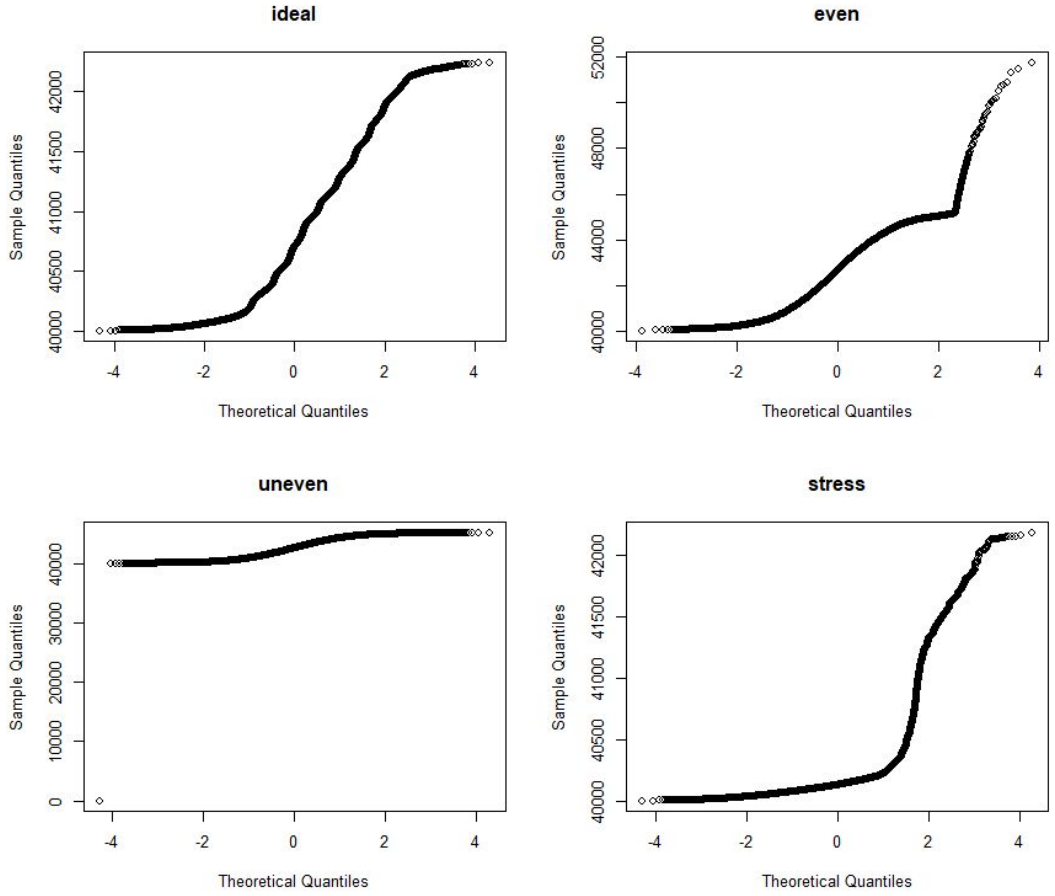

Figure 1: Q-Q Plot of job completion times across the various loads

The below qq plots visualize the distribution of the job completion times across the various loads. Not that we expected the data to be normally distributed, as we know the ratio of job completions vary immensely from workload to workload. The above plots are simply to give an indication of how the job completion times are distributed, and as we can see it is very skewed (which is to be expected). Also as expected, the ideal workload produces the most (close to) normal distribution compared to all other workloads.

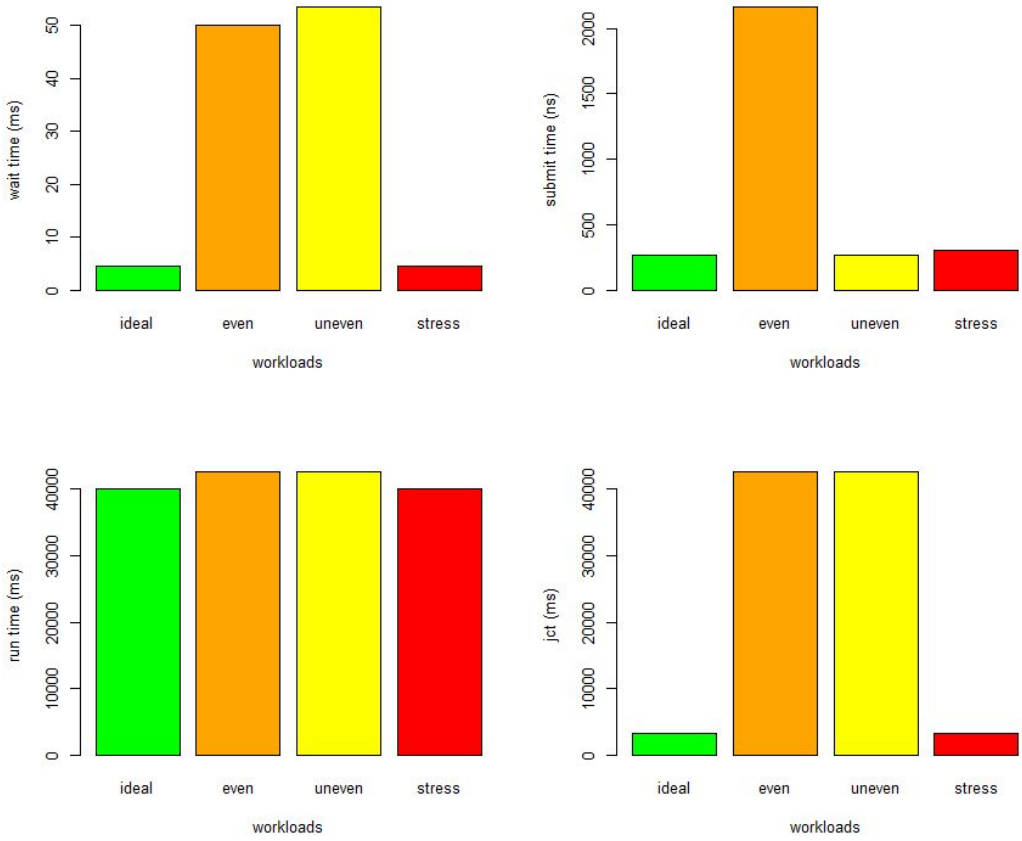

Figure 2: Barplots of job metrics

The above bar-plots depict the differences in the average job submission, wait, run and completion times across the various loads. They are created from the sample data that can also be found in the Appendix.

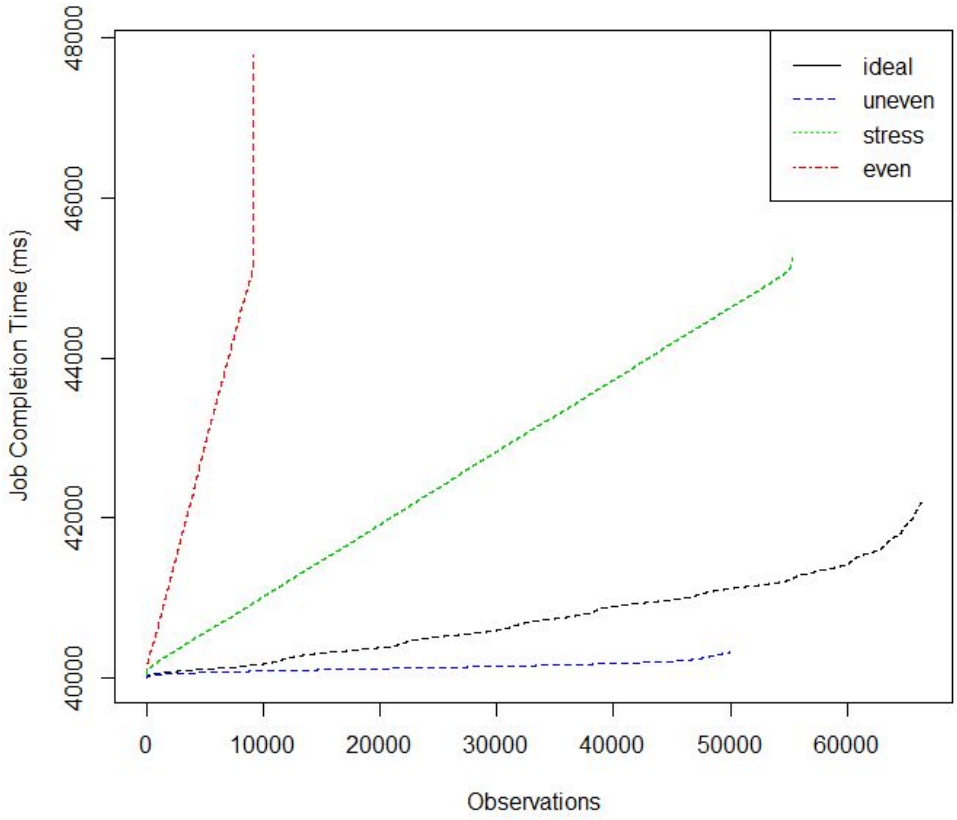

Figure 3: Job completion times in ascending order

This graph is generated from job completion times from all workloads in ascending order. It shows how the value for the various job completion times evolve (aggregate), up until the execution is over. The data sample from each workload differ in size/length, as the experiments ended based on a time-limit. Under ideal load, we have far more observations (70k+) which results in the longer (black) line.

V. Discussion

Job Completion Time is defined as the sum of Submit Time, Wait Time and Run Time. Theoretically, the most important factor of the three is the Wait Time which we try to keep low. Through the workloads applied, the most interesting observation is the unexpectedly high value of the submit time of the even workload. This makes sense in a way that jobs are randomly distributed across the clusters. That randomness adds an overhead which is embedded in the Submit Time calculation.

Another remarkable and high praising on the VGS remark, is that in the stress workload, wait time is in the magnitude of ideal workload. That remark shows how efficient the system is when is comes to evening the load under stress conditions. However, even and uneven workloads introduce randomness to the duration of the jobs which guarantees that the duration is either the same with the rest of the workloads or even longer (max 5 seconds longer).

More detailed experiments could have been conducted but we emphasized on JCT. We think it's the only final metric that matters and everything else comes in second thought. Based on that belief, we were satisfied with the results and didn't need to do any further experimentation.

VI. Conclusion

Coming up with a Virtual Grid System which fully achieves performance, availability, fault tolerance and its consequences (replication, consistency, synchronization) at the same time, is impossible. Our goal was to implement a robust system against local faults so they would not result into a system failure. During that challenge, we had to make proper design decisions which deal with the various trade-offs in a kinda fair way without scaling in favor or against one specific DS property. We kept scalability in mind while we opted for quick job completion at the same. Our approach is simple. Carry out the job fast or assign it to someone else (another cluster) who can. In that way, we aim for low Job Completion Time which was also the metric of our experiments.

The beauty of our implementation is that many of its major components are scalable. Number of Clusters, number of processing nodes and number of GS Groups can scale. The propagation ring concept allows scalability in higher level. However, performance may be severely affected because it takes much more time to propagate the load while some GS Nodes are overcrowded with jobs. In that case, it is better to divide the ring into two subrings and in that way you actually have two independent sub-VGSs. Fault tolerance protocol is quite smart because it uses, by design, replicas which do not need synchronization. Synchronization would have added quite an overhead. Instead, we use shared data structures and killing mechanisms for redundant instances. Due to the emphasis on a low JCT, on some occasions, we observed that some clusters are 100% utilized while others had a low utilization percentage. One could argue that this is not evening the load and we would agree. Our response on this argument would be that it is not bad that some clusters are 100% utilized while others are not. That is actually good. It is only when the load is over 100% that we need to even that load by migrating jobs to others. If we evened the load at all load percentages that would mean redundant migration overhead and long JCTs.

Experimenting and testing, due to local machine reasons, was quite challenging because we tested our implementation via a simulation. However, it yielded some bugs which they were quite hard to fix due to

the complexity and concurrency of the running threads. The main problem was trying to replicate the bug in order to corner it. Distributed Testing offers so many areas for experimentation. However, we focused on the metric that is the most important for the client: Job Execution Time. That metric can reveal many secrets about where the most delays are introduced and tackle those areas. Fortunately, the results were satisfying which reflected our meticulous job and focus on the VGS architecture. We believe that a good initial design requires much fewer modifications and tweaks in the future, instead of a redesign necessity.

To sum up with, we are confident about the efficiency of the system and its ability to even the load when needed. Our emphasis on the client's needs is more than obvious and we believe that we came up with an innovative system which serves the requirements in the most efficient way. That emphasis on client's needs and the scalability ability of the system are our most valuable arguments in our effort to convince WantDS.BV to adopt our idea and hire us for their venture.

Appendix

Architecture overview

This figure below depicts the major components of the VGS and some basic functionality.

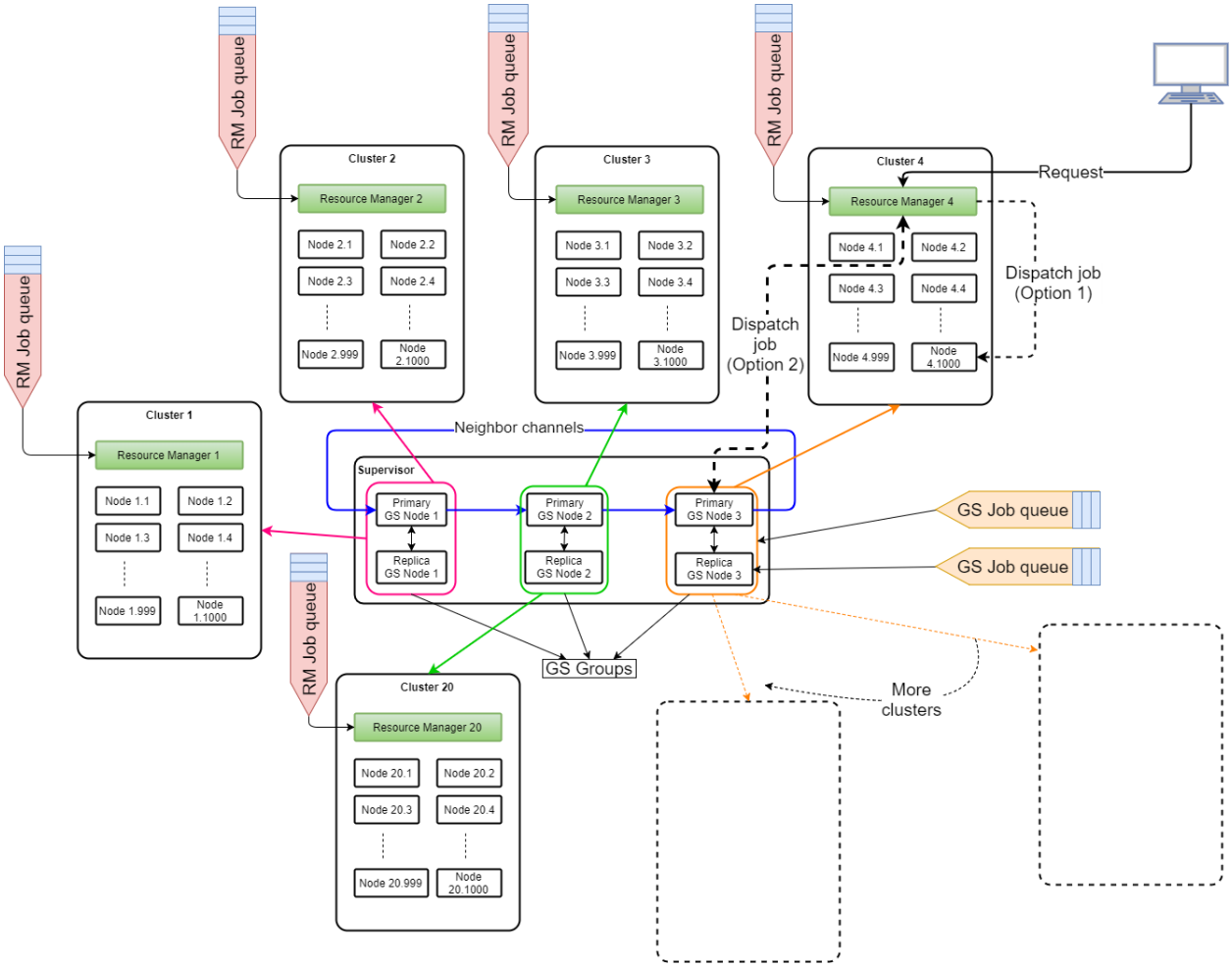

Figure 1.0 *Architecture overview*

GS Node Fault Tolerance

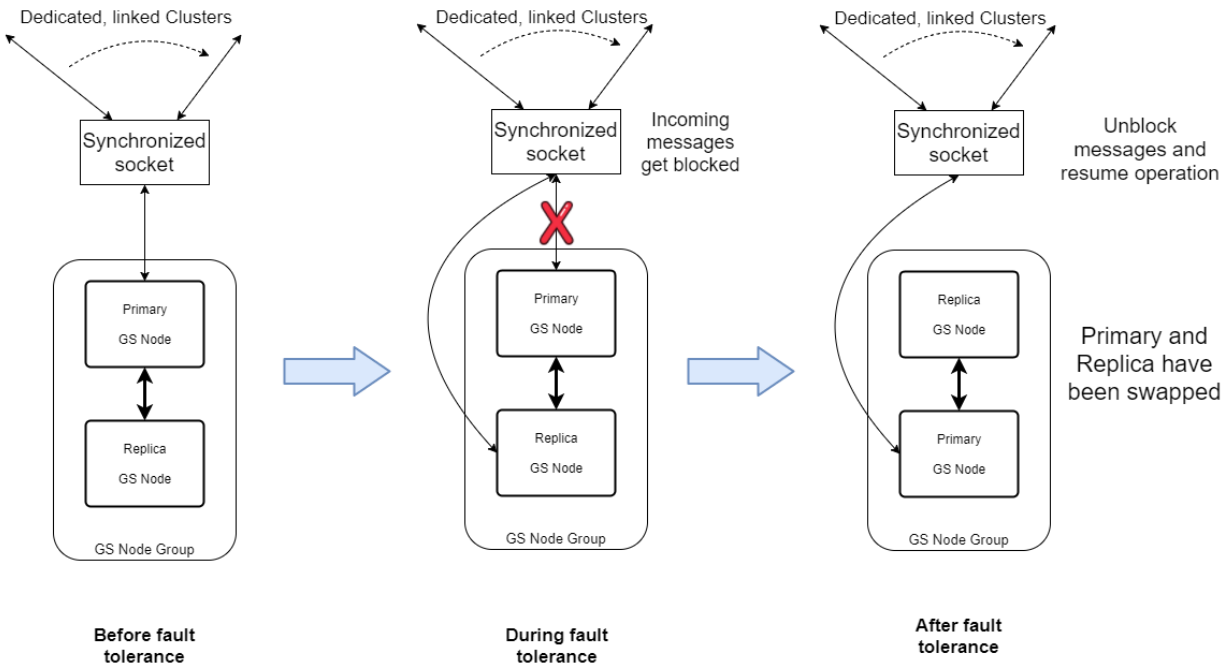

Figure 1.1 *Architecture overview*

Evening the Load

This picture depicts the protocol of evening the load by migrating jobs from one GS Node to another

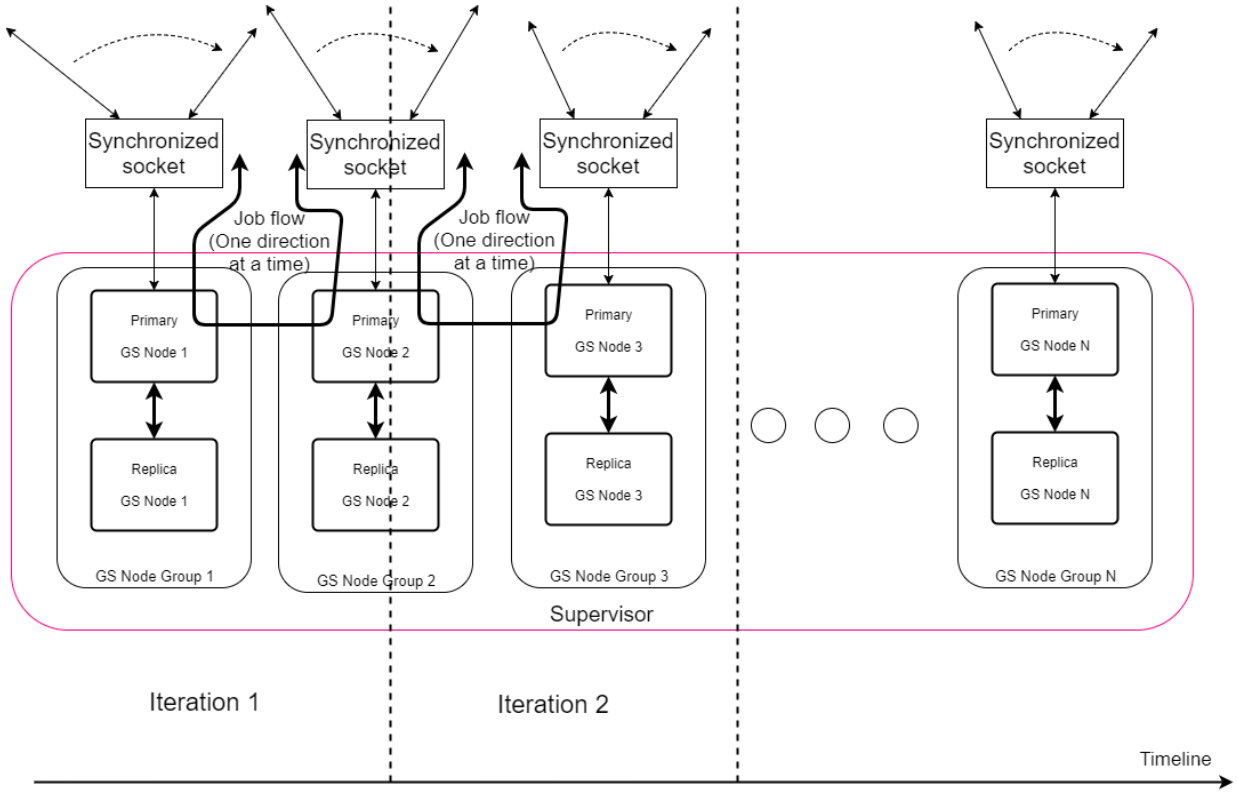

Figure 1.2 *Even the load*

Data Procurement and Workload Information

In the tables below we emphasize the various system settings and other configurations made to the system to extract the data we needed. We also provide system specifications for the machine on which experiments were executed.

Machine Specifications	
CPU	2.2 GHz Intel Core i7
Memory	16 GB 1600 MHz DDR3
Storage	Macintosh SSD 256
Graphics	Intel Iris Pro 1536 MB

Averages				
Load	Submit (ns)	Wait (ms)	Run (ms)	Completion (ms)
Ideal	264	4.54	40051	3270
Even	2165	50	42613	42663
Uneven	271	53	42602	42602
Stress	301	4.51	40052	3250

Workloads	
Load	Details
Ideal	Simulates an ideal load on the clusters, meaning that all clusters get the same amount of jobs in a given time.
Stress	This option is most useful because it stresses the system by applying the given ratio. The ratio is a positive integer and expresses the ratio of the numbers of jobs arriving at the most and least loaded cluster. That number should most of the times be at least 5.
Even	Randomly distributes jobs across the clusters. The distribution nearly follows the uniform distribution.
Uneven	The opposite of the previous option. It bombards the randomly picked cluster with 5 times more jobs than the randomly picked least loaded cluster.

System Scales and Component Settings

	Component Settings					
Workload	Ratio	Clusters	GS Nodes	Processing Nodes	Jobs	Job Duration
Stress	20	8	4	1000	10000	40 sec
Ideal	N/A	8	4	1000	10000	40 sec
Even	20	8	4	1000	10000	40 + Random (0,5]
Uneven	20	8	4	1000	10000	40 + Random (0,5]

Standard Workload Format Subset

The metrics we use are a small subset of the Standard Workload Format[1] from the Grid Workloads Archive.

Identification fields	
Time- and status- related	

ID	SWF	GWF	Info
1	Job Number	JobID	A counter field, starting from 1
2	Submit Time	SubmitTime	In seconds. The earliest time the log refers to is zero, and is the submittal time of the first job. The lines in the log are sorted by ascending submittal times. It makes sense for jobs to also be numbered in this order.
3	Wait Time	WaitTime	In seconds. The difference between the job's submit time and the time at which it actually began to run. Naturally, this is only relevant to real logs, not to models.
4	Run Time	RunTime	In seconds. The wall clock time the job was running (end time minus start time). We decided to use "wait time" and "runtime" instead of the equivalent "start time" and "end time" because they are directly attributable to the scheduler and application, and are more suitable for models where only the run time is relevant. Note that when values are rounded to an integral number of seconds (as often happens in logs) a run time of 0 is possible and means the job ran for less than 0.5 seconds. On the other hand it is permissible to use floating point values for time fields.

[1] http://gwa.ewi.tudelft.nl/grid-workload-format/

Time Allocation Table

	Weeks 2-3	Weeks 4-5	Weeks 6-7
Think	11h	16h	5h
Develop	13h	21h	6h
Experiment	-	4h	2h
Analyze	-	-	4h
Write	4h	10h	26h
Waste	5h	10h	9h
Total	**146h**		

Github Repository

github.com/michelcojocaru/vgsTeam08

References

[1] https://www.cis.upenn.edu/~lee/07cis505/Lec/lec-ch1-DistSys-v4.pdf