IN4391: Large lab exercise A: Distributed Virtual Grid Simulator

Ir. Yong Guo
Delft University of Technology
Yong.Guo@tudelft.nl

Dr. ir. Alexandru Iosup
Delft University of Technology
A.Iosup@tudelft.nl

Michael de Jong
Delft University of Technology
M.deJong-2@student.tudelft.nl
1314793

April 1, 2013

1 Abstract

Clusters or grids of computers allow their users to run massively parallel jobs. One interesting and active area of research is to explore the possibilities of joining multiple grids together for better reliability and throughput. These grids are expensive to build and operate. So in order to investigate how one can couple multiple grids together is best simulated in a smaller environment. For this purpose I present a system called Distributed Virtual Grid Simulator (*DVGS*) which simulates an environment where multiple grids are managed by a group of distributed schedulers.

DVGS is comprised of an arbitrary number of resource managers: nodes which represent a grid capable of running jobs of a certain duration, and which have a certain maximum capacity of simultaneous running jobs. *DVGS* employs at least one scheduler which is capable of redistributing jobs to other resource managers if a resource manager is fully utilized and cannot accept an offered job. Multiple schedulers can be run in tandem to provide fail-over in case the master scheduler goes down.

Simulations of *DVGS* show that it allows single grids to ‘accept‘ more jobs than they could do without *DVGS*. This is due to the fact that *DVGS* allows grids to offload their exceeding load to other grids with less utilization.

2 Introduction

2.1 Problem statement

Clusters and grids are comprised of many tens to thousands of computers which typically use a network to communicate with each other. These computers can execute many tasks or jobs of varying duration and load in parallel. Systems like this are particularly interesting when a problem can be reduced to many smaller problems which is each solvable by a single computer. Solving all required subproblems with a single machine might take very long so grids can be used to distribute these jobs across multiple machine. This allows their users to solve these large and complex problems in a relatively short timespan.

Typically a grid is managed by a single node called a resource manager. This node accepts incoming job offers and distributes them to idle nodes in the grid. Resource managers are a bottleneck for each grid and a single point of failures (*SPOF*). From a user's perspective this means that if the resource manager crashes, other nodes in the same the grid will not be able to process their jobs.

Another issue is utilization. Every grid has a certain amount of resources which allow it to run up to a certain number of simultaneous jobs. If all the resources of a grid are fully occupied with jobs, the resource manager is faced with a problem. It can either decline the offer, or store it in a queue. In the latter case the resource manager has a serious problem if it crashes because it will loose the jobs stored in the queue.

2.2 Solutions

There have been several attempts to link multiple grids together. But the most notable system known to me, is a system called Condor Flocking [1] is built on top of an existing grid middleware solution for single grids. In this system gateways impersonate a resource within a grid, and simply relay the jobs to another Condor grid. This was done to ensure that the original Condor middleware did not have to be modified.

DVGS attempts to provide a fully distributed, very fault-tolerant layer between the grids allowing them to offload excessive load to other less utilized grids in the network. More details on the design and implementation can be found in section **??**.

2.3 Structure of this article

The remainder of this article is structured as followed: In section 3 I will explain some of the requirements that have been established to which a system has to adhere. In section 4 I will discuss the design and implementation of the created distributed system *DVGS*. In section 5 I will present some of the experiments and results that were done with the *DVGS* system. In section 6 I will elaborate on the experiment results and properties of the constructed *DVGS* system. Finally in section 7 I will briefly state my conclusion of this report. Additionally in appendix A, a description of the time it took to construct the *DVGS* system can be found.

[1]D.H.J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne, "A Worldwide Flock of Condors: Load Sharing among Workstation Clusters, Future Generation Computer Systems, Vol. 12, pp. 53-65, 1996.

3 Background on Application

3.1 Previous work

The original Virtual Grid Scheduler (*VGS*) system on which *DVGS* is based, is a Java application which can simulate a grid handling jobs. Every offered job has a set duration. When a resource manager receives a job, it starts a timer with the duration of the job and decrements the available capacity with 1. When the timer has elapsed, it increments the available capacity with 1, significy that the job has ended. This simple mechanism allows us to run very large simulations of running jobs on grids. *DVGS* employs the same mechanism to simulate grids.

3.2 Requirements

- **Operation** Jobs can be submitted to any resource manager in the distributed system. A resource manager can put submitted jobs in a local job queue or pass them on to a scheduler node. The scheduler node can in turn send it to a resource manager, which has to accept it and put the job in its local queue.
- **Fault-tolerance** The distributed system must be fault tolerant. This means that the system should be able to cope with a crashing node. To facilitate this, at least two schedulers must keep track of the state of the entire distributed system. In case one goes down the system can continue to operate.
- **Scalability** The system should be able to simulate a system consisting of 20 resource managers representing grids with a maximum capacity of 1,000 concurrent jobs, and 5 schedulers.

4 System Design

4.1 Messaging

One of the suggested means of messaging between nodes in the distributed system was Java's Remote Method Invocation (RMI). Each node can call a remote method with a Message object and a Address object, describing the sender. The sender can either wait for the method call to complete, which in essence signifies an acknowledgement, or ignore the result and continue onwards right after sending the Message.

4.2 Topology

DVGS is built with a high degree of fault-tolerance in mind. I wanted the system to continue working as long as their was at least one scheduler operational. Additionally I wanted the system to recognize new schedulers and resource managers trying to join the distributed system. In order to facilitate this, every node in the distributed system periodically scans a predefined range of IP addresses and ports. If it can establish a connection with an unknown node, it will send its currently known Topology. This object consists of all known nodes and their roles (resource manager / scheduler) in the distributed system. When an object receives a Topology object it will merge the data from that Topology with its own, and try to connect to nodes it previously did not know about.

This simple mechanism allows nodes to quickly discover each other and exchange information about what role they play in the network. If a node crashes, it will eventually (during the next scan) or immediately (if a Message could not be delivered to it) be dropped from the Topology object, which is then broadcast to the rest of the network. This same mechanism allows new nodes (both schedulers and resource managers) to join the grid when they become available.

4.3 Chain of command

Schedulers are responsible for redirecting jobs to under utilized grids, each managed by a resource manager. In *DVGS* schedulers are ordered by age. The oldest scheduler acts as the master scheduler and is responsible for assigning jobs to grids. In case the master scheduler crashes, it is removed from the Topology and the then oldest scheduler becomes the master scheduler. In this linear ordering each scheduler has a backup with the exception of the youngest scheduler. If a new scheduler is detected using the mechanism from the previous subsection, it is automatically added to the end of the scheduler ordering.

4.4 Replication & Fault-tolerance

Replication is a very important requirement in this system. If a user submits a job and the system acknowledges that submission, the user should not have to worry that the system will lose it when one of the nodes crashes. In order to ensure this, the system employs a system of replication to achieve a very high fault-tolerance.

When a resource manager wants to send a message to the schedulers it first sends it to the master scheduler. Before this message is acknowledged the scheduler will forward the message to its immediate backup, which will do the same until there is a scheduler which has no immediate backup. The backup will then acknowledge the message it received from its predecessor. Eventually the master scheduler will receive an acknowledgement from its backup that it has replicated the message, upon which the master can acknowledge the original message it received from the resource manager.

In case the master scheduler has crashed, the resource managers message will time-out and will attempt to resend the message to the next scheduler in line. In case a non-master scheduler has crashed, its predecessor will receive a time-out, and attempt to replicate the message to the next backup in line. This basic scheme allows this system to replicate jobs to all schedulers in the grid, even if some schedulers have gone offline.

4.5 Scheduling

The current implementation of *DVGS* employs a very simple mechanism of scheduling jobs. When a user offers a job to a resource manager, it will check locally if it has enough capacity to run this job. If it does it will send an ACCEPT message to the scheduler which includes the job information, and an index of the current cluster utilization level. If it does not have enough capacity it will send a RESCHEDULE message to the scheduler with that same information. If it finished a job the resource manager sends a FINISHED message to the scheduler also with this information. If the scheduler receives a RESCHEDULE message it will update the cluster utilization level index for that cluster, and figure out the least utilized cluster at the moment (with spare capacity). The job is then forwarded to that resource manager. If none of the resource managers have any spare capacity the scheduler will hold the job in a queue until one of the resource managers has spare capacity.

The master scheduler keeps and maintains a model of the entire grid. This model describes what jobs are at which nodes in the grid and in what state (PENDING, RUNNING, FINISHED). This model is replicated to each of the schedulers as described in the previous subsection. In case the master scheduler crashes, its immediate backup will automatically take over once it notices that the master scheduler has gone offline.

5 Experimental Results

Because the development of *DVGS* took considerable time for a one-person team. The time left to experiment with this system was limited.

5.1 Experimental setup

DVGS is lightweight enough to run simulations of 25 nodes (5 schedulers and 20 resource managers) on a single MacBook Air 2011 (1.8 Ghz i7, 4 GB memory). Nor the DAS cluster, nor Amazon's EC2 platform was used to run the experiments. Since the system can easily run on a single laptop, this was preferred to running the experiments on multiple physical machines since experiments are easier to monitor and execute. The system was however also tested on multiple physical machines (2 laptops and a dual-core desktop computer) as well to demonstrate that it works.

DVGS employs Java's Remote Method Invocation (RMI) to call methods on remote machines with locally specified parameters. On top of RMI a small layer of code has been built which abstracts it into a messaging system with acknowledgements. More details on this have been described in the *System Design* section of this report.

The system is packaged into .jar files using Maven's package command. This .jar file is then distributed to all the physical machines which will be running this software. It can be configured to either run (multiple nodes) locally on a single machine, or a single node on a single machine connected to a network of other machines.

5.2 Experiments

To test if the system meets the requirements specified earlier, we ran a simulation of 5 schedulers and 20 resource managers. These nodes were started and as soon as each node had found the other 24 nodes the simulation started. 10,000 jobs were submitted to a single resource manager in batches of 500 jobs each.

5.2.1 Consistency and fault-tolerance

In order to test consistency and fault-tolerance, I ran the simulation and killed one or more nodes in the grid (both schedulers and resource managers), and checked that still all 10,000 jobs were eventually marked as finished by one of the remaining schedulers. In each case the system was able verify that all 10,000 jobs had finished.

5.2.2 Scalability

The requirements stated a minimum scale of 5 schedulers and 20 resource managers with 10,000 submitted jobs. This is easily achievable but the current system has not been tested beyond this scale. My intuition is that the system will be able to cope with bigger scales that the specified scale, but at the cost of eventual consistency. In other words it will take longer before the system has become consistent after a change has occurred.

5.2.3 Performance

When jobs are offered to a resource manager and this resource manager is capable of handling the jobs itself, only ACCEPT messages are sent to the schedulers. If the resource manager is not capable of handling the jobs it sends RESCHEDULE messages to the schedulers. There are considerably more messages sent in the latter case. This message overhead becomes increasingly bigger the closer the entire grid comes to full utilization. When all resource managers are fully utilized they will automatically forward all incoming jobs to the schedulers which has to store them locally and replicate them to the all backup schedulers. Then when one or more resource managers have finished some of their jobs, the schedulers have to redirect any pending jobs to them and replicate this new information to the backup schedulers. This scenario can generate a substantial set of messages which slow down the redistribution of jobs.

6 Discussion

6.1 Main findings & tradeoffs

- There's an inherent issue with the volume of messaging with regards to replication of information and dealing with an highly utilized grid. *DVGS* replicates all messages and state of the grid to all schedulers. The requirements listed 5 schedulers as a minimum, but 2 or 3 schedulers are enough to recover from a crash and keep running. This could greatly reduce the number of messages to be sent.

- Users are allowed to submit jobs but the resource managers don't have any mechanism in place to protect them from an overload of job submissions. *DVGS* allows users to batch jobs before submitting them to one of the resource managers, but this is not enough to protect the system from slowing down.

6.2 Recommendations

DVGS is a system which meets the requirements of WantDS. Since new nodes can be added or removed to and from the grid, it is advisable to only use 2 or 3 schedulers at any given time to be able to cope with a crashing scheduler.

7 Conclusion

DVGS is a system which has met all the stated requirements and also allows the operator of the system to add schedulers and resource managers to the grid at run-time. The advanced fault-tolerance ensures that no job is lost when a scheduler or resource manager crashes.

8 Appendix A

- Total-time: 61h
- Think-time: 6h
- Development-time: 40h
- Experiment-time: 4h
- Analysis-time: 2h
- Wasted-time: 1h
- Write-time: 8h