Virtual Grid System Final Report

Mengmeng Ye
4407326
M.Ye-1@student.tudelft.nl
Changliang Luo
4501357
C.Luo@student.tudelft.nl

Abstract

As the technology and economy grows, the need for Grid Computing System grows as well. Grid Computing System can integrate heterogenous resources, in other word, it can make full use of whatever resources you have. In addition, it is much more stable than a centralized system (if well designed). On the Request of WantDS BV, we design this multi-cluster grid computing system. The system is expected to have some degree of fault-tolerance.

1. Introduction

The needs of computing is growing. In the area of scientific research, business or even daily life, we consume a lot of "computing". To meet the needs of our society, building centralized single super computers is an expensive choice. Grid Computing System is an alternative. The distributed nature of Grid Computing System has pros and cons: it has heavy cost in communication between nodes in the system, relatively low performance, but it also have advantages like scalability and robust.

A Grid Computing System consists of multiple subsystems, those systems are relatively autonomous, they communicate with other subsystem by some protocols. To cooperate with other subsystems, a subsystem needs to frequently send and receive message, this means heavy cost of communication. Also, compared with centralized system, the Grid Computing System are somewhat "lack organization", it lack a central supervisor who knows the big picture of the whole system, so Grid Computing System are usually not able to act in an global optimal way, that means lower performance.

However, the distributed nature also brings benefits. First of all, the Grid Computing System is scalable. If a person uses a centralized system, e.g. a single powerful super computer, one day he wants to double computing power, he has to build a new super computer with double computing power. That may means 4 times cost and sometimes even impossible (we can't build a single super computer with infinite computing power). However, in a Grid Computing System, the only thing one person needs to do is add more computers into his system, the cost grows nearly linearly, and a Grid Computing System is also more stable. While a power failure can take down a centralized system, the Grid Computing System may work well even half of the hosts in the system are suffering power failure.

Grid Computing System has pros and cons, should we pursue the benefits at the cost of those drawbacks? To find out the answer, the WantDS BV ask us to decide a Grid Computing System simulator. Examining the performance of this simulator may give some light to the answer.

2. System Overview

WantDS BV have some requirement for our Grid Computing System Simulator.
The Grid Computing System Simulator is consist of 4 kinds of hosts: Grid Scheduler (GS), Resource Manager(RM), Node, and Job sender (can be seen as Client).
A Node is the basic working unit in Grid Computing System. Its function is very simple: get a job, do it and report to its Resource Manager after job accomplished. In our simulator, the job is simply waiting a certain period of time. The Nodes don't need to cooperate with each other.
A Resource Manager is a leader of a cluster, the cluster is a set of Nodes.
Grid Scheduler is the leader of Resource Managers.
A Job Sender can send the job to any Resource Manager. When a Resource Manager Receive a Job from Job Sender, it can either assign it to the Nodes in its cluster, or offload it to Grid Scheduler if the workload in this cluster is to high. When a Grid Scheduler receive a job from a Resource Manager, it will assign it to the cluster with lowers workload, in this case, the Resource Manager can't offload the job to Grid Scheduler anymore.

3. System Design

3.1. Basic Design

On the request of WantDS BV, the System should have at least 5 Grid Scheduler, 20 Resource Manager and 1000 Nodes. Theoretically we need 1025 hosts to run this system, however, what WantDS BV asks us to build is a simulator, we don't need 1025 real machines to run such system. In our design, we do the simulation in following way: we run every member (Node/Resource Manager/Grid Scheduler) of the System as an independent RMI server binding to a name and a port, e.g. rmi://45.23.21.55:5001/GS1. When we want to deploy them to different machine, we only need to reset the IP. Every RMI server is running in a separate thread (can also be run in separate process), so calling function of a RMI server won't interve another RMI server.

3.2. Architecture

Our System consists of multiple subsystems: every 50 Nodes and 1 Resource Manager will form a cluster, every 4 cluster share a same Grid Scheduler.
For test purpose, Job sender can send job to whatever Resource Manager it wants.
The Architecture is shown in figure 1.
The Resource Managers maintain some information of its Nodes, e.g. URL, id, status (busy or idle).
Grid Scheduler maintains information of all Resource Manager (not only its own Resource Manager). You may notice that here is some redundancy, we will talk about it later.

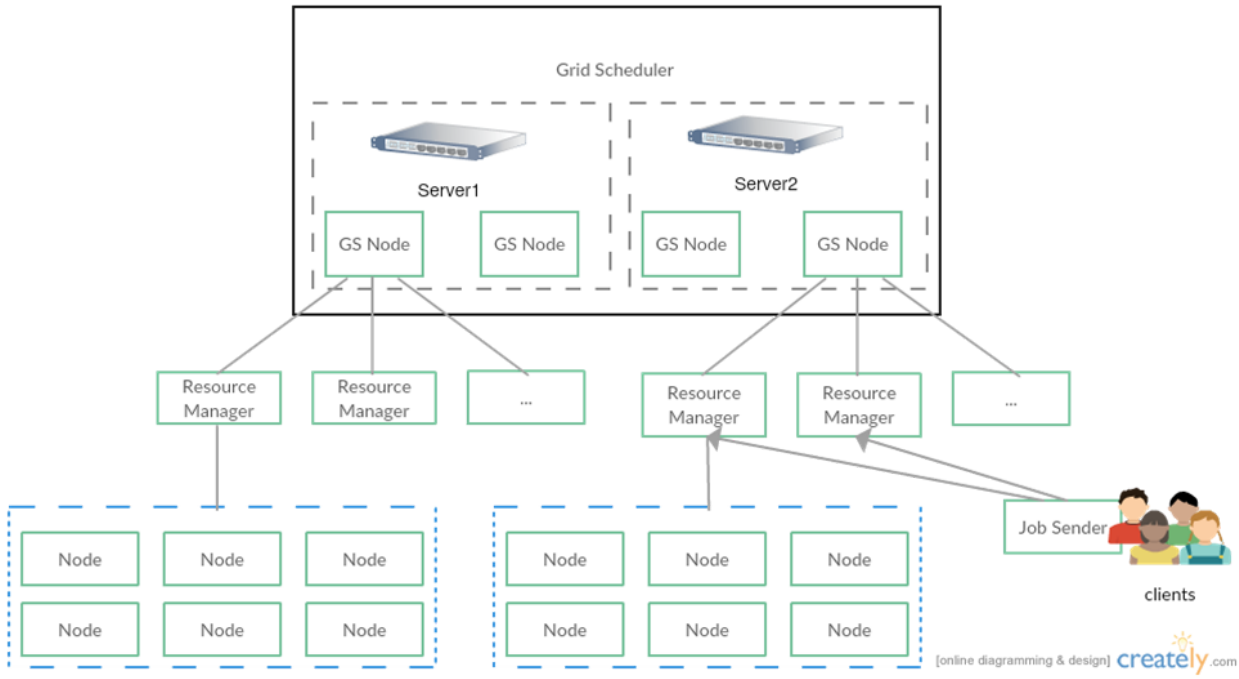

Figure 1 Architecture

3.3. Load Balance

In real scenario, it is common that one cluster has too many jobs while other clusters have no job to do. To prevent this situation and make good use of the resources in Grid Computing System, it is necessary to design a load balance mechanism which can transfer some jobs from a busy Resource Manager to idle ones.

Our load balance mechanism is below:

If a Resource Manager Receive a job and it realizes that the workload of this cluster exceeds threshold (can be specified), it will offload it to its Grid Scheduler. The GS maintain the metric of the workload of every RM. Every time the GS receive a job and wants to assign it to a RM, it can actively check the workload of all its RM. GS will pick up the RM with least workload and assign job to it.

Here comes a very interesting topic. There is a possible scenario: let's say RM A is the RM with least workload, suddenly a great amount of jobs go to its GS and GS check work load of RM and find that RM A is the most idle one and assign to RM A, as a consequence, RM A become very very busy, if the workload is huge enough, it can even knock down RM A.

This scenario can be prevented by our system. Because even a great amount of job offloaded to a GS, it will only assign one job per turn to a RM, and after assigning that job, the RM will immediately update its metric; next time the GS assign job to this RM, it will check the workload of this RM again. In addition, the job can be assigned to a RM only if the previous job has been assigned (assigning job occupy the main thread of GS, that means it disable concurrently assigning job), so this mechanism guarantees that every time a GS want to assign a job, the workload of that RM (in GS's view) is up to date. When this RM (used to be idle) is full, the GS will turn to another RM rather than continuing assign jobs to it. There will be a experiment for load balance in the next section.

However, I admit that to prevent the scenario mentioned above, we don't allow GS to concurrently assigning job. We actually prevent such scenario at the cost of some performance, but we think it worth doing so.

3.4. Consistency

Our system maintains some degree of consistency between RM and GS. The RM will periodically upload its status to all GS, including the list of its Nodes and a list of job, so if a RM crashes the GS can send its previous status to another RM and the new RM will easily take its place. Here comes a trade off, if we upload frequently, we can maintain a strong consistency but we use great amount of resources to upload and receive those message. If we upload it not so frequently, the consistency is weak. We choose to maintain a relatively weak consistency on the assumption that the RM is not so likely to crash.

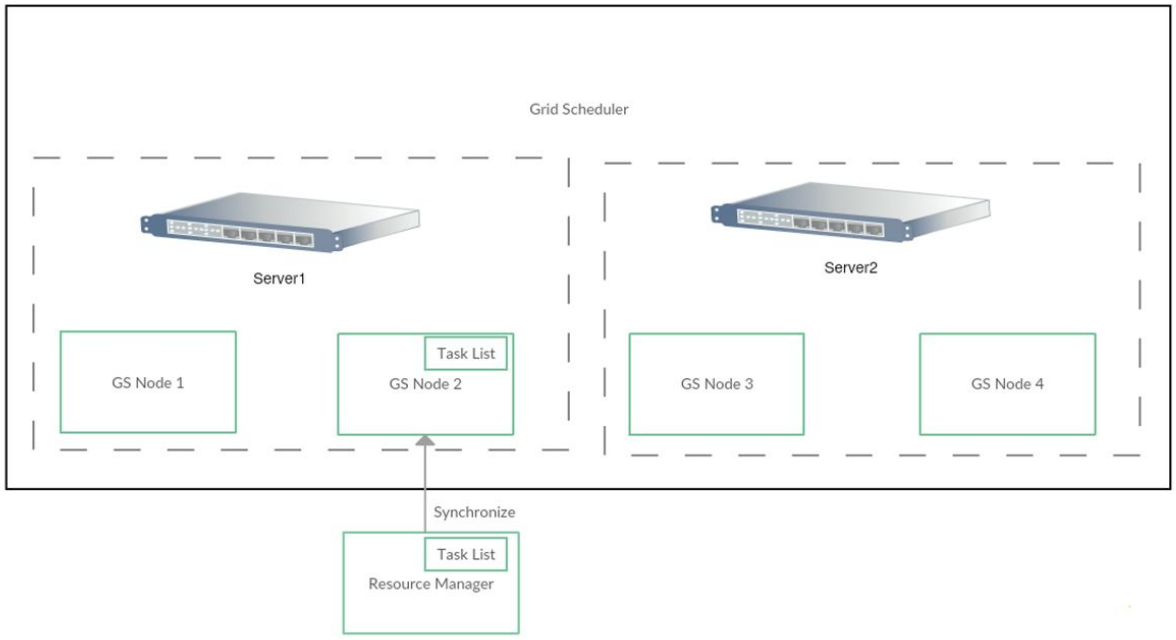

Figure 2 Consistency between Grid Scheduler and Resource Manager

3.5. Fault Tolerance

Our System has some degree of Fault Tolerance. It has mechanism to deal with multiple GS/RM crash accident.

Every GS will periodically check whether other GS are alive (there is a remote function in GS, it can be called using GS's url, this function will return a "true"), if there is no response over 2 times, that GS will be seen as crash.

Once a GS crashes, all other GS will take action. The GS with smallest ID will take action immediately, and the GS with second smallest ID will wait for a period of time (e.g. 5 seconds), and the third smallest

will wait longer (let's say 10 seconds) etc. Once a GS has taken action, it will send a Message to other GS to cancel the pending action, so that there won't be duplicated action.
The GS who takes action will also send a Message to RMs, the message telling which GS is dead, and every RM will check whether the crashed GS is its own GS, if so, it will register to another GS.
Our System also has fault tolerance for RM crash accident. In this Case, its GS will move the Nodes (as well as its job list) in its cluster to another RM. The way GS detecting RM crash is the same with detecting GS crash.
But there are some drawbacks for this mechanism. First, we detect GS/RM crash using timing, here comes a trade off: if we set the time interval between two detections very small, we will be aware of a crash very quickly, but that creates a lot of traffic in network, it is a heavy burden; but if we set the time interval very big, it will take us a long time to realize a GS/RM has crashed. Second, there is such a scenario: a GS/RM hasn't crashed, but just too busy to reply the crash detecting, so it is seen to be dead. As a consequence all its RM/Nodes move to others. Currently we don't have mechanism allows a GS/RM to reclaim its RM/Node. However,we admit that this mechanism is necessary.

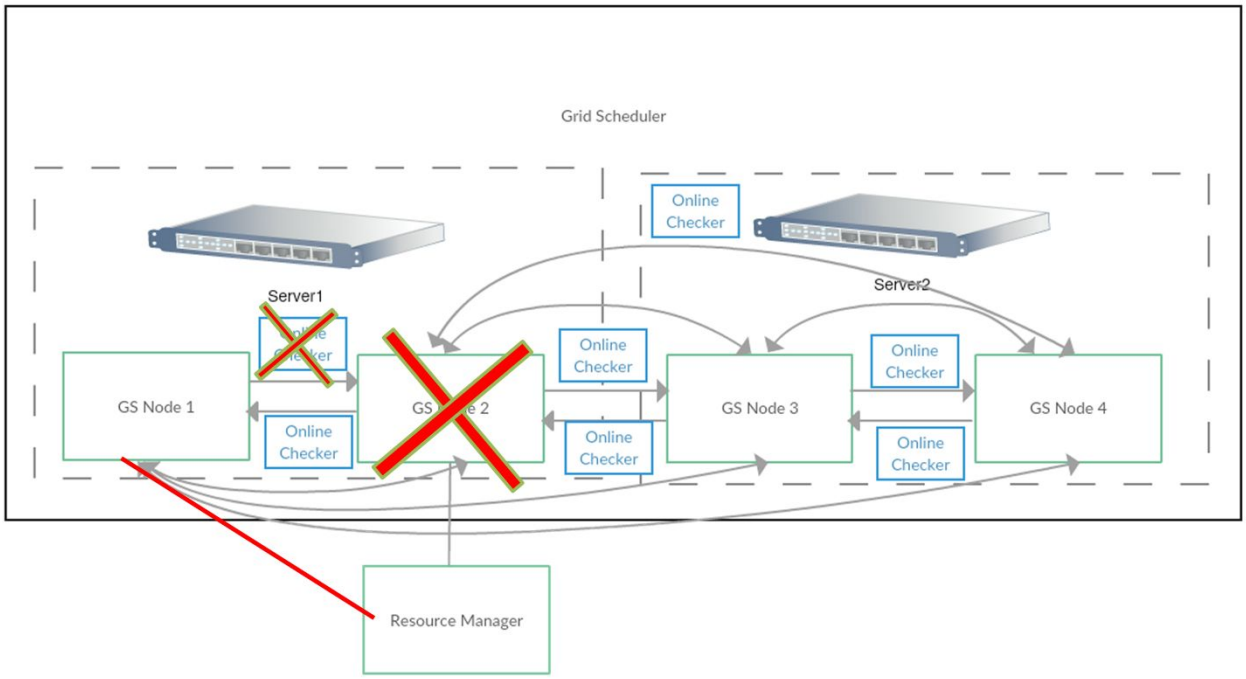

Figure 3 GS crash tolerance

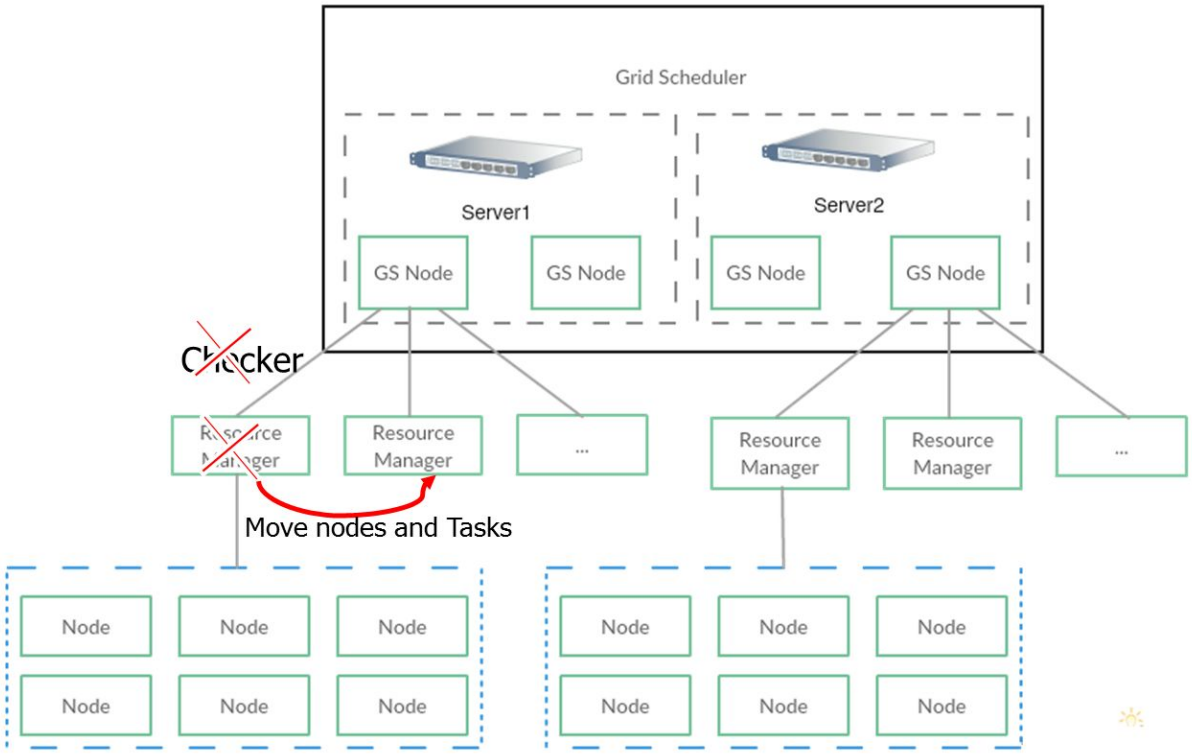

Figure 4 RM crash

3.6.Scalability

We expect our system has some degree of scalability. We also expect that the Grid Scheduler will be the bottleneck of scalability.

As our system grows, e.g. we make number of Node, Resource Manager and Grid Scheduler as 4 times as now. The burden of Node and Resource Manager won't increase as the size of the system grows. However, the burden of Grid Scheduler will grows, they are now responsible to check more Grid Schedulers are they still alive.

A solution for the bottleneck is organizing Grid Scheduler in different groups. For example, Grid Scheduler 0~4 are put in a same group while 5~9 are put in another group. A grid scheduler is only responsible for detecting the crash of its own group member. However, since we now have only 5 Grid Scheduler, we haven't do the grouping yet.

4. Experiment

After building our system, we do some experiment to measure the performance of our system, the experiment are mainly about Load Balance,Fault Tolerance and reliability.

4.1.Load Balance

In this experiment, we keep sending a set of jobs (enough to fill up 2 clusters) to a random Resource Manager, since a single cluster can't take so many jobs simultaneously, it should offload part of the jobs to its GridScheduler.
If our Load Balance mechanism is well designed, eventually the workload of all clusters should be even, otherwise there will be extremely busy and extremely idle cluster.
We record the workload of every cluster every 5 seconds and the experiment lasts for over 5000 seconds. Since we have 20 clusters, they are too many to be shown in single graph, we randomly pick up 5 of them and demonstrates in the figure 5

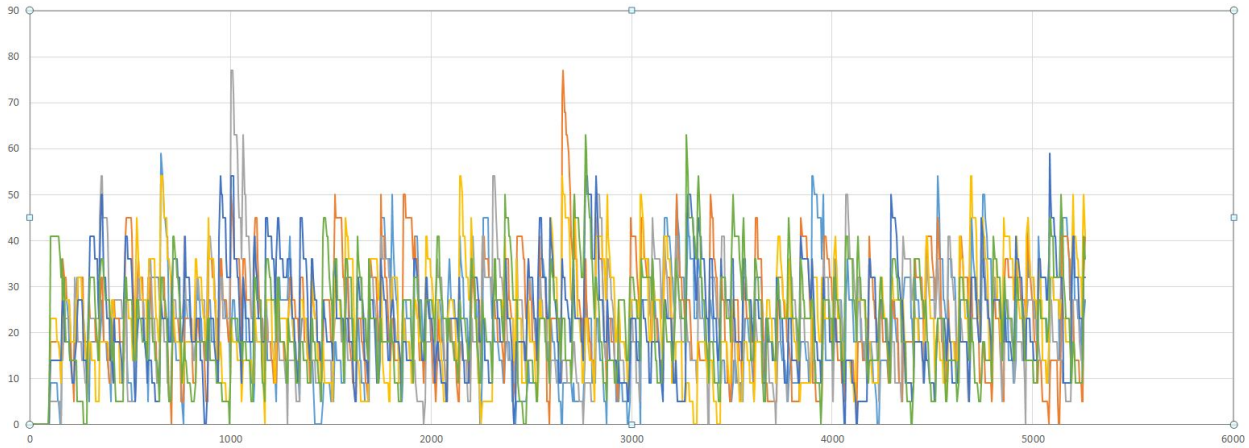

Figure 5 workload of different cluster

Different cluster is represented by different color. You may notice that the workload of different clusters are in general even. There are some sharp "bump" in the graph, that is due to a sudden arrival of many jobs from a job sender. The Resource Manager will quickly offload such jobs to its grid scheduler, as a consequence, the workload decrease sharply.
Just as our expectation in previous section, since we require GS to assign jobs one by one, not concurrently, we prevent the situation that the GS suddenly assign all jobs to an idle cluster and hence make it extremly busy. But this method is not efficient if the network latency is very high, every time you need to assign a job, it takes you a few seconds to request the workload of every cluster.
This topics should be left for WantDS BV, adopting this method or an alternatives depends on what they value most, efficiency or consistency?

4.2. Fault Tolerance

In our design, our system have some degree of fault tolerance for GS/RM crashes, in this experiment we test the fault tolerance of our system by randomly take out a GS/RM and test how much time the system needs to be aware of the crash.
We tested how long does it take to detect that GS/RM is crashed respectively as in the Figure 6 and Figure 7. The average crash detection time of RM is 5.94s, and that time for GS is 9.43. The detection time is a bit long because we set the crash checker to run a check every several seconds, rather than running all the time.

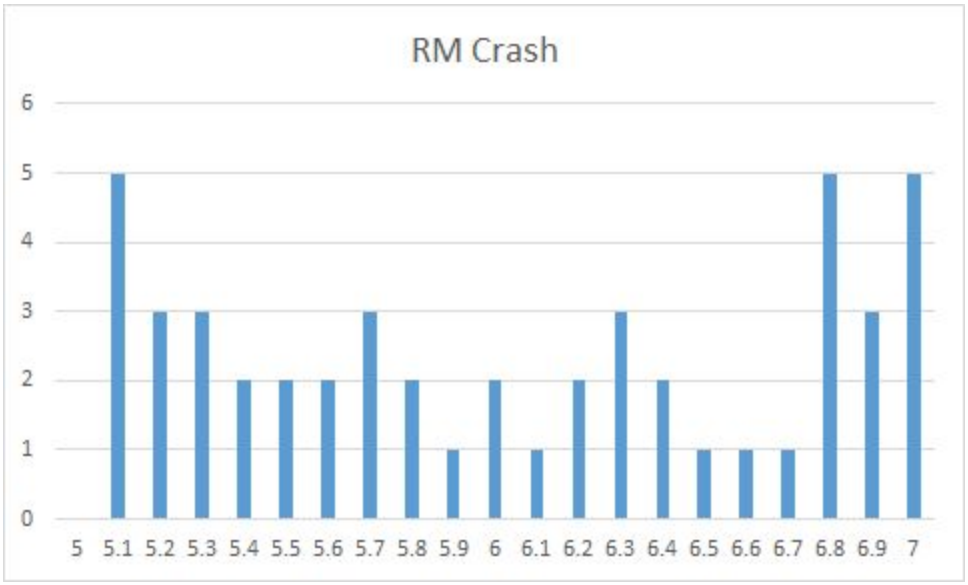

Figure 6 Detection time of RM Crash

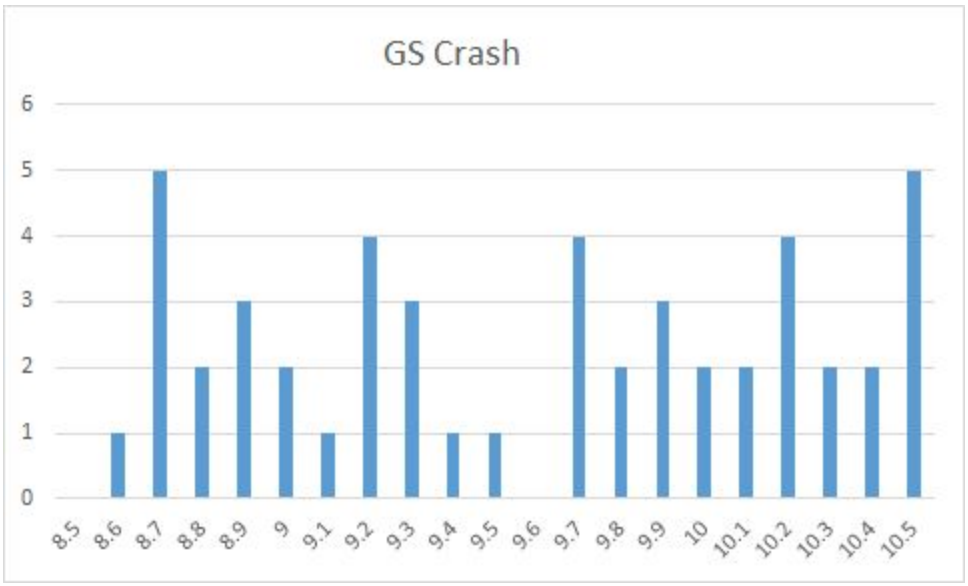

Figure 7 Detection time of GS Crash

4.3. Scalability

We fail to find out the maximum scalability of our system. Before we meet the bottleneck of our system, we meet the bottleneck of our virtual server (we use 2 remote virtual server with 2G memory), when we increase the size of our system 4 times, our servers are on the edge.

5. Conclusion

In this report we introduce the design of our Grid Computing system. It is a system with a few subsystems (cluster). When jobs arrive at the Resource Manager, it can choose either assign them to its own Nodes or offload it to Grid Scheduler. A Grid Scheduler has the duty to balance the workload over clusters. In our design we consider the scenario of multiple GS/RM crash, we include a fault tolerance mechanism to deal with such crash.

After building the system, we do a few experiment to test the performance of the system. It turns out that it can balance the workload over clusters well; it can also deal with GS/RM crash. But we fail to test the maximum scalability of our system. We expect that there should be a bottleneck but fail to find out the maximum scalability.

6. Appendix

6.1. Appendix A: Time Allocation

Mengmeng Ye

Think time: 5h
Group discussion: 10h
Implement time: 43h
Deploy time: 14h
Test time: 4h
Write time: 6h
Total: 82h

Changliang Luo

Think time: 5h
Group discussion: 10h
Implement time:45h
Deploy time: 18h
Test time: 4h
Write time: 6h
Total: 88h

6.2. Appendix B: Open Source

We made our project open source online, the url is:
https://github.com/ymm1993326/GridComputing

7. Reference

Tanenbaum, Andrew S., and Maarten Van Steen. Distributed systems: principles and paradigms. Vol. 2.

Englewood Cliffs: Prentice hall, 2002.