# IN4391 Distributed Computing Systems Virtual Grid Distributed System

Santiago Natalio Conde Camacho
4486153
sacondecamacho@gmail.com

Ferdy Moon Soo Beekmans
1327755
ferdymoonsoobeekmans@gmail.com

Course instructors:
Dr. Ir. A. Iosup
Ir. Alexey Ilyushkin

**Abstract**

**With an increasing use of distributed computing systems, the study of their performance is becoming a top priority. Motivated by this, a distributed architecture of the Virtual Grid System is developed on request for WantDS BV. The design of the system considers two approaches, the ring topology and the fully connected topology, with the focus on fault tolerance, scalability and load balance. The chosen approach was the fully connected one. Experiments are conducted to measure the performance of the system, especially when dealing with multiple users simultaneously. From the result of these experiments, the conclusion can be made that the requirements imposed by WantDS BV are met by the chosen design.**

## 1 Introduction

With the explosion of the data, the industry has set its sight on distributed computing systems for their data processing problems. However, this field has several technical challenges and problems to solve. For instance, what happens when part of the system that was executing some work crashes? How to distribute workload across the system in a reasonable balanced manner? To tackle this scenario, WantDS BV has the goal of designing and implementing a distributed simulator of a multi-cluster system and study its feasibility.

The distributed system designed is formed by several grids of independent clusters of nodes that carry out user workloads, consisting on sequential stateless asynchronous jobs of a fixed duration. Each grid is monitored and controlled by one or more grid scheduler (GS).

This document describes the process of designing, developing and evaluating such system. Section 2 details the requirements of the proposed Virtual Grid System (VGS). Section 3 analyzes the considered design approaches and compares their respective merits and deficiencies. In section 4 the most important details of the implementation are briefly explained. Subsequently, section 5 shows the results of the experiments conducted to evaluate and assess the system. Finally, the designed and implemented system and its trade-offs are discussed in Section 6, followed by a conclusion that contains the verdict whether the system is feasible or not.

## 2 Virtual Grid System

The Virtual Grid System (VGS) is a tool to monitor how a simulated distributed multi-grid system performs conducting user workloads. Each cluster contains two types of resources: processors (nodes that carry out jobs) and a resource manager (RM) which controls and monitors the cluster. For grid operation, such a set of clusters needs a grid scheduler (GS), which enables load balancing across the clusters and oversees the correct execution of the jobs. Users send their workload (jobs) to the cluster of their choice. Depending on whether the cluster has resources available to attend the request or not, the job will be added to the cluster's waiting queue or delegated to a grid scheduler for its later allocation, either in the same or in a different cluster. If an RM receives a job from a GS, it has to accept it, even if that means adding it to the local waiting queue. Once the job is finished, the RM will send the result to the requester user.

The system designed consists on 5 grid schedulers and 20 clusters containing 1,000 nodes each. The minimum workload the system is able to support is 10,000 jobs, for simplicity purposes each job demands only one processor for its execution. When a job arrives at a cluster it records the cluster's identifier, in case that the job is reallocated to a different cluster it will trace the additional identifiers of the clusters where it passes.

The system is resilient to single failures. That is, the system is able to operate normally even if there is a failure in a GS or in an RM, the user is not be aware of these failures, her workload is carried out transparently. Lastly, the system handles with load imbalance. The ratio of the jobs arriving at the most and least loaded cluster the system is able to tolerate is 5:1.

# 3 System design

In this chapter, the design of the system is explained. The focus of section 3.1 will be an overview of the first approach implemented, the ring topology. Here, a detailed analysis of this architecture and its trade-offs is presented. However, due to the issues and complexities described below, it was discarded and not finished. Subsequently, in section 3.2 there is an analysis of the second approach, the implemented one, where components have full connectivity. Lastly, in section 3.3 we discuss the additional features implemented in the system.

## 3.1 Ring topology

The proposed architecture, shown in figure 1, consists on a logical ring of GSs and evenly divided groups of RMs. Each group of RMs is monitored by one GS (blue lines on the figure), being monitored means that the GS oversees the execution of the workloads conducted at those clusters. Thus, periodically, RMs ping the monitoring GS indicating they are alive and what current load of the cluster is. Additionally, each group of RMs is backed up by the next GS in the ring (red lines on the figure) thus, RMs also ping the back-up GS periodically. The role of the back-up GS is to supervise the RMs and take action in case of failure. Apart from that, every GS exchange information about the clusters under its supervision with the next and previous GS of the logical ring (black lines).

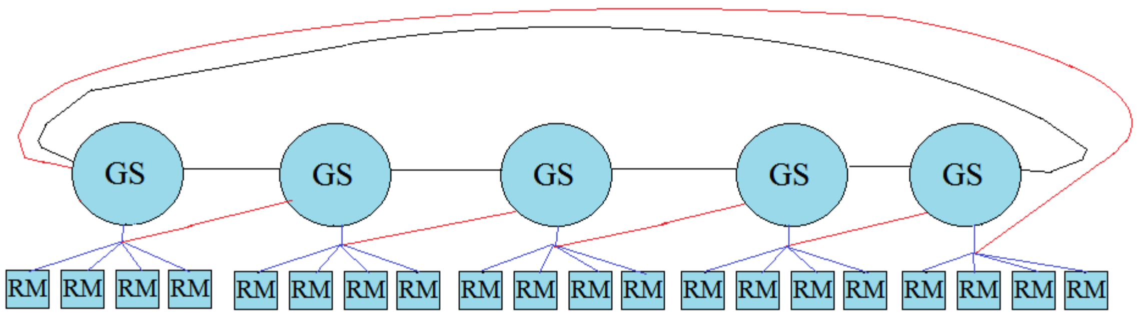

Figure 1: Ring topology

### 3.1.1 Initialization

Since the topology is fixed, the components need to know in advance with whom they are going to be connected to, which makes initialization hard and complex.

### 3.1.2 Workflow

When a job request arrives at an RM this will decide either to queue or to offload the request based on the current load. In case that there are resources available to attend the request, the RM will first send the job to both, monitoring and back-up GS for replication. Once the replication requests have been acknowledged, the user will be notified that the her job has been accepted and that it will be queued for execution. Otherwise, the job is sent to the monitoring GS, which will handle the allocation of the job.

The job is executed by the first node which becomes available. Upon completion of the execution, the RM informs the monitoring and back-up GS that the monitoring resources allocated for that job request can be released. A diagram of the workflow is displayed on the figure 2 below.

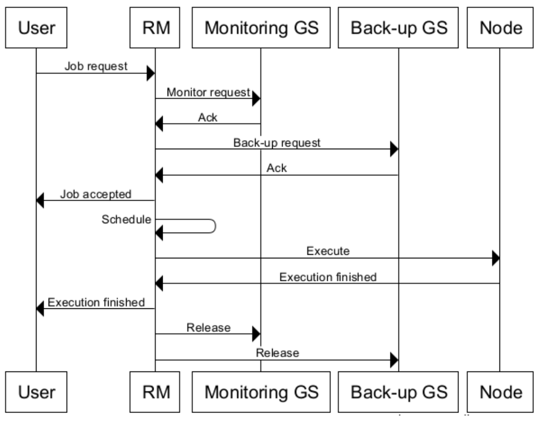

Figure 2: Workflow in the ring topology

**Fault recovery** RMs and GSs may fail at any time and restart, failures are discovered due to ping time-out. When a monitoring GS notices that one of the RMs under its supervision has failed, it will inform the back-up GS to stop monitoring the crashed RM. Immediately after that, it will reallocate the jobs which were executing in the failed RM to a new cluster. Once the new RM receives the workload from the GS, it will proceed with the replication and scheduling as in the normal workflow. On the other hand, when a back-up GS detects the failure of the monitoring GS, it will promote itself to become the monitor of the clusters supervised by the crashed GS. Subsequently, the RM will connect to the next GS on the ring, next to the failed one, as backup GS. The action flow in the case of back-up GS crash is rather similar, the RM will connect to the next GS in the ring as back-up.

This strategy of fault recovery may seem robust, however, it is likely to trigger a failure of the whole system due to a domino effect in the fault recovery. The reason behind that is that all the workload monitored by a failed GS is entirely supported by the back-up. If the latter is already heavy loaded, it will probably fail as well, triggering a cascade failure. This scenario is also possible in the case of an RM failure, the GS monitoring the new selected cluster will assume all the incoming workload. Without a protocol to re-balance the topology when a component resumes its activity, all RMs will eventually being monitored by a single GS.

**Load balance** As a response to the pings, GSs receive the load of the clusters they monitor, this load is then propagated around the ring in both directions. In this way, when a GS receives a job offloaded by an RM it can easily choose a new cluster trying to balance the load of the whole system. The way of selecting the new RM is by using an inverted weighed random selection with the load of each cluster.

### 3.1.3 Scalability

Adding a new GS will only imply connecting the new component to another two GSs and two RMs (one as main monitoring and one as back-up). Similarly, an extra RM will just ping a main GS and a back-up GS. Thus the topology does not present message overload and scales with $O(1)$.

### 3.1.4 Issues

In this topology each component has dependencies on specific other components, making the initialization of the system difficult. Moreover, when failures occur, the topology of the system changes. In order to re-balance the new topology, a protocol for joining the system would be needed, which adds significant more complexity. Despite the fact that this topology was designed to be resilient against failures, domino failures may kill the whole system, as discussed above. Considering these issues, the described approach does not seem adequate.

## 3.2 Fully connected topology

Given the issues of the ring topology described above, we chose to implement a second approach, described in the remainder of this subsection. To circumvent said issues, in this solution GSs are fully connected among themselves and RMs ping all GSs present in the system. A diagram of the topology is shown in figure 3 below.

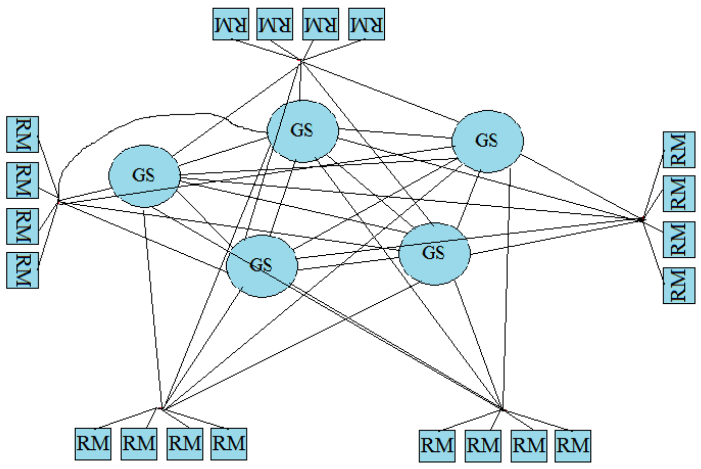

Figure 3: Fully connected topology

### 3.2.1 Initialization

All the components have a list of URLs of the whole system, once started they begin to ping periodically each other. Each RM is ready to serve requests once it is connected to at least two GSs.

### 3.2.2 Workflow

To account for the changed topology, some changes have been made regarding the workflow. Once a job requests arrives at an RM, a monitoring and back-up GS are chosen randomly among all the GSs of the system. When an RM needs to offload a job request, it also selects randomly the GS to offload the job to. Similarly to the ring topology, the chosen GS will allocate the job to an RM which will take care of the execution.

**Fault recovery** Failure detection is detected in the same manner as in the ring topology. However, because the monitoring and back-up selection is made randomly for each job request, it is unlikely to trigger a domino failure, making the system more robust. Also, when a crashed component resumes its activity, it will simply notify the rest of the system so it can be selected again for monitoring/back-up (in the case of a GS) or processing job requests in the case of an RM. Another improvement compared to the ring topology is the tolerance against multiples crashes. Single failures of GSs are handled in the same way than in the ring topology, however the way of handling RM crashes is slightly different. In this setup, once the monitoring GS notices that an RM has crashed, it will first check whether the back-up GS is still alive, once an acknowledge is received, it will reallocate the jobs in another RM. Yet, since there are no fixed GS assigned to any RM, the monitoring GS will also monitor these jobs during their execution in the new selected RM. Thus, the new RM will only request a new back-up GS before scheduling the jobs. Another important difference to note is that the selection of the new RM is done for each of the jobs present in the failed RM, since the new RM is chosen by using an inverted weighed random selection, it is extremely unlikely that a single RM receives all the workload of the failed one, reducing the possibility of cascade failure.

In the case of a simultaneous crash of an RM and its monitoring GS, the back-up will promote itself to become the monitoring GS of the workload of the failed RM. After that, the recovery process is identical to the case of a single RM crash. Similarly, if the back-up GS and RM crash at the same time, the recovery process is conducted by the survival monitoring GS, it will reallocate the jobs in new RMs, which will select new back-ups. Lastly, if both GSs fail simultaneously, the RM will select a new monitor and back-up before continuing its activity. Workflow diagrams of the recovery processes for the scenarios explained above can be found in Appendix B.

**Load balance** The load balancing process is exactly the same than in the ring topology, however, because every RM pings every GS, the retransmission of the messages with load among GSs is not needed.

#### 3.2.3 Scalability

This topology scales with $O(|GS| \cdot |GS| - 1 + |RM| \cdot |GS|) = O(n^2)$. Where $|GS|$ and $|RM|$ denote the number of GSs and RMs respectively. The first term $|GS| \cdot |GS| - 1$ is the number of messages among GSs, whereas the second term $|RM| \cdot |GS|$ indicates the number of messages between RM and GS. This scalability factor may become an issue, however, taking into account that the size of the pings is extremely small, as they just contain the load of the clusters, the total overload is not too high.

### 3.3 Additional features

As explained above, the system designed is able to tolerate failures in at most two nodes at the same time. Actually, it can tolerate multiple combinations of failures in GS/RM as long as there are components nodes to support the total workload of the system. Going beyond what it was previously said, the system has the ability to deal simultaneously with diverse users since job requests include a user identifier. As it will be shown in the experiments section, the different workloads are treated evenly, in this way the system offers an equitative service.

## 4 Implementation

This section will shortly review the main implementation decisions.

### 4.1 Future

To implement asynchronous execution Futures were used. A Future is a placeholder object for the result of a computation that can be run asynchronously. The result will become available upon completion. This abstraction is more similar to objects and methods than the threading alternative, which is closer to the hardware implementation. However, the easy use of asynchronous operations can be treacherous since it hides the possibility of deadlocks. Asynchronous programming is somewhat equivalent to program distributed computing without network failures.

### 4.2 Repository

The different components discover and communicate with each other through a Repository. Every time an entity is referenced or pinged, its state is updated. Interaction with other components is done by invoking callbacks on the respective entity, these can be either targeted or randomly selected using a selector. The callback will then be executed on the component resulted of the selection. In the case that the entity is not available, the Repository will keep retrying until no entities are available and report the failure. Every time a change in the state of another component is observed, a hook is triggered executing all the callbacks registered in that hook. This mechanism is used to implement fault recovery.

### 4.3 Scheduling

Executing the jobs is implemented by scheduling their completion after the amount of milliseconds that the job is meant to run for. By using a scheduler instead of using *Thread.sleep* threads are not blocked, significantly lowering the number of context switches and improving the performance of the system.

## 5 Experiments

### 5.1 Experiments setup

Apart from the local setup used for developing and debugging purposes, the whole system was deployed in Amazon Web Services platform. To do so, 4 EC2 t2.micro instances were used. These type of instances have one virtual CPU with a maximum frequency of 3.3 GHz. One of the instances hosts 5 GSs, RMs are divided in two groups of 10 clusters, each group running in different instance. The last virtual machine acts as user, it generates several user processes which request several amounts of jobs of fixed duration to several clusters. This was the setup used at the interview were the capabilities of the system were presented. All the code used for the project and the experiments can be found on GitHub[1].

[1] https://github.com/fydio/vgs

## 5.2 Experiments

### 5.2.1 Fairness of the system

When dealing with multiple users at the same time one of the most important metrics is the capabilities that each of those users perceive from the system. In the system designed, one of the factors that concern the users is the job time completion, especially when the load on the system is high. The system will be considered *fair* if the users see similar job completion times in average when they send workload of the same characteristics. To test that, the job completion times for 4 users each of them sending a workload of 4,000 jobs that required 2s each are analyzed. The results are displayed in table 1.

| User | Av.completion time in s | Time variance in ms |
|---|---|---|
| 1 | 2.21 | 30.7 |
| 2 | 2.15 | 11.6 |
| 3 | 2.10 | 9 |
| 4 | 2.21 | 16.2 |

Table 1: 2 Job completion times per user

As it is possible to appreciate, the different users receive a similar completion time, the variance is not high, which shows that most values are around the average.

### 5.2.2 Limit to scalability

As noted in the design section, the burden of handling a quadratically increasing amount of messages might be a bottleneck. To test this in isolation, different system sizes were used on a single machine to mitigate the effect of a possible higher transmission time if the components were deployed on a different machines. Another option could be to test with every cluster and GS on their own machine. Due to the size of the experiment, this was deemed infeasible. The workloads were kept small, but over capacity of the system, such that the computing power would minimally impact the results. Each user submitted 100 jobs of 1s to every RM. Increasingly large test setups are used to see the decline in job completion time. The used system sizes are listed in table 2

| Users | Resource managers | Grid Schedulers |
|---|---|---|
| 5 | 20 | 5 |
| 10 | 40 | 10 |
| 20 | 80 | 20 |
| 40 | 160 | 40 |

Table 2: Test setup sizes.

Figure 4 shows the completion times (time between request and response) in the different virtual network sizes. As expected the, larger networks on average handle requests quicker. This can be explained because not all jobs are submitted exactly at the same time, leaving a bigger buffer of resources to offload for the initial jobs. As the network grows bigger, some outlier jobs take longer to complete. A possible explanation is that some jobs are waiting for other operations to complete due to locking mechanisms used. It is not likely that this problem is caused just by handling ping requests. This operation has a low computational cost and the vast majority of jobs is still handled quickly.

From this experiment we can conclude that the system performs well at least with double the amount of components the system was intended for. Above that, the system can still handle most jobs well but performance degrades.

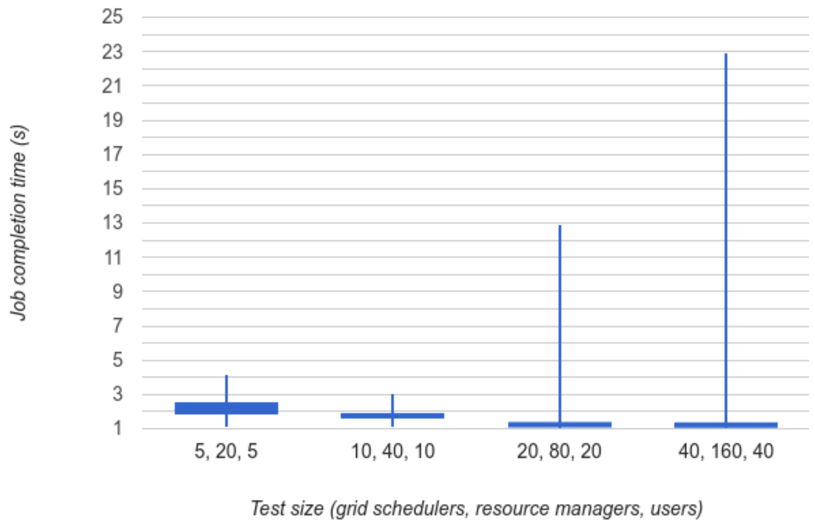

Figure 4: Job completion times

#### 5.2.3 Fault tolerance

To verify that system can successfully recover from the different failure scenarios we created specific simulations were some components fail at a certain point. The logs displaying the result are available in appendix B.5

# 6 Discussion

In this chapter the trade-offs inherent in the design of the system are discussed. Based on the results described in the section above a verdict will provided deciding whether WantDS BV should use a distributed system as the one designed or not.

## 6.1 Topology trade-offs

As discussed on the section 3, two different topologies were analyzed. The ring topology was initially chosen because of the robustness of that kind of networks [2]. Another merit for this topology is the low message exchange in the whole system, making it easily scalable. Additional components could be added in the hypothetical situation that the system needs to conduct a higher workload not covered by current capabilities. However, as analyzed in section 3.1 the topology is rather sensitive to cascading failures. This sensibility is inherent to the design of the topology, even in a completely balanced system where every cluster has a maximum load of 50% over the total capacity, the failure of one RM will cause the overload of the new selected cluster. Nevertheless, since the implementation of the first approach was not completed there is no empiric confirmation regarding this issue. The likely possibility of cascading failures and, the need of a complex protocol to coordinate how components join the system on initialization and after a failure were enough reasons to discard this design. On the other hand, as discussed earlier, the fully connected topology is easy to initialize, the Repository solution, presented in section 4, facilitates the discovery and starting process as components only need to know the URLs of the rest of the system. Thus, there is no need of a protocol to rearrange the topology when components come back to life after a failure, they will simply contact again the entities on the repository and their status will be automatically updated. However, in massive systems with thousands of clusters this process may become slower since they have to contact all the entities. The random selection of a GS for monitoring and back-up mitigates the impact of overloading a specific component reducing the risk of domino failure. A negative consequence of the fault recovery scheme is that if an RM fails after all the jobs have been submitted, it will remain idle for the rest of the execution even if the other clusters are full. However, in realistic scenarios it is sensible to assume that the arrival patterns will not consist on only one burst of job requests but more continuous over the time.

## 6.2 Scalability

As advanced on the section 3, the scalability of the system is $O(n^2)$, nonetheless, as the sizes of the ping messages and the job requests are not very high, the system is able to work properly under the required conditions. It is expected that if there is a change on the workload characteristics or the number of components scales several orders of magnitude the system will not be able to provide the service it is intended to. However, as explained in the experiments section the system respond well to workloads above the requirements, leaving a considerable margin in case that WantDS needs to upgrade its system.

### 6.3 Load balance

GSs load balance the system in two different ways: first, when a job is offloaded from an RM, the receptor GS will select the new RM using an inverted weighed random selection with the load of the clusters. This means that GSs will keep the system balanced while dealing with offloaded jobs. The second way is when an RM crashes, the GS in charge of its recovery will also select the new RMs using the aforementioned selector, thus, the system is also balanced while dealing with failures. There are other possible load balancing policies that were not included on the system for the sake of simplicity, such as an active load balance initiated by GSs. In that scenario, GSs will periodically take action and reallocate jobs from heavy loaded clusters to lightly ones. This new possibility, even if not implemented, is thought to be more effective in terms of load balance. However, new issues would arise with this policy, for instance, establishing the threshold for the reallocation to start, whether clusters should suspend their activity while the reallocating process takes place or, which GS initiates the load balance.

## 7 Conclusion

In this report, the design of the Virtual Grid System (VGS) is presented. First, the two considered topologies, the ring topology and the fully connected topology are analyzed and compared regarding fault tolerance, load balance and scalability. Due to the sensitivity to cascading failures and the need of a complex protocol to coordinate initialization and how components reconnect after a failure, the ring topology was discarded. Next, the main implementation decisions are explained. After that, the conducted experiments are described. The focus of the experiments is on the fairness of the system while dealing with multiple users at the same time, fault tolerance, and scalability limits. Finally, the trade-offs and drawbacks of the design are discussed.

After performing the experiments, we can conclude that the system is resilient to two simultaneous failures. While the number of messages in the system is high, the system is able to provide the functionality it is intended to, with a considerable margin before scalability starts becoming an issue. When dealing with multiple users at the same time the system treats users evenly, offering a similar average completion execution time per job to every user. Lastly, the load of the system is balanced by the GSs when a job is offloaded or needs to be reallocated in a new cluster due to a failure. The conclusion of this report is that WantDS should use a distributed system as defined in this report, since it meets all of their requirements on fault tolerance, scalability and load balance.

# Appendices

# Appendix A Time allocation per group member

## A.1 Santiago Natalio Conde Camacho

| Type | Hours |
|---|---|
| `think-time` | 5 |
| `dev-time` | 25 |
| `xp-time` | 12 |
| `analysis-time` | 10 |
| `write-time` | 42 |
| `wasted-time` | 12 |
| `total-time` | 106 |

## A.2 Ferdy Moon So Beekmens

| Type | Hours |
|---|---|
| `think-time` | 8 |
| `dev-time` | 47 |
| `xp-time` | 13 |
| `analysis-time` | 21 |
| `write-time` | 5 |
| `wasted-time` | 25 |
| `total-time` | 114 |

# Appendix B Fault recovery workflows

## B.1 Fully Connected Topology - RM Crash recovery

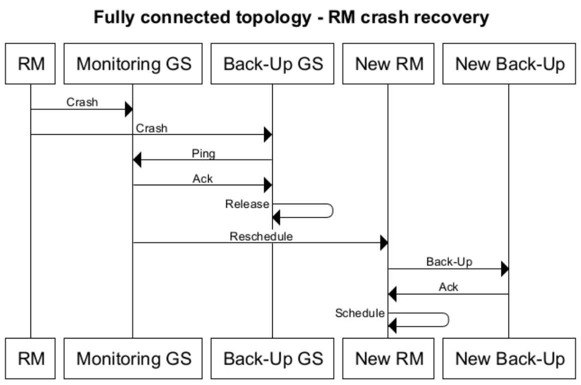

Figure 5: Fully connected topology - RM Crash recovery

## B.2 Fully Connected Topology - RM & Monitor Crash recovery

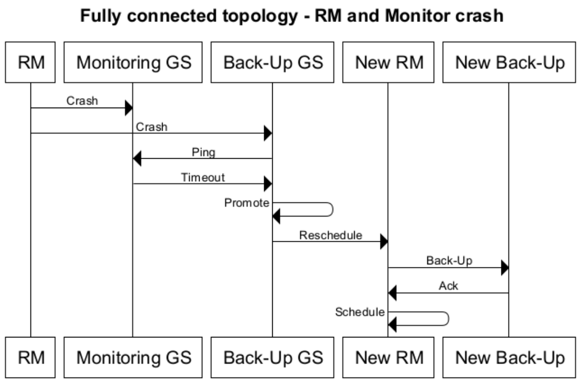

Figure 6: Fully connected topology - RM and Monitor Crash recovery

## B.3 Fully Connected Topology - RM & Back-up Crash recovery

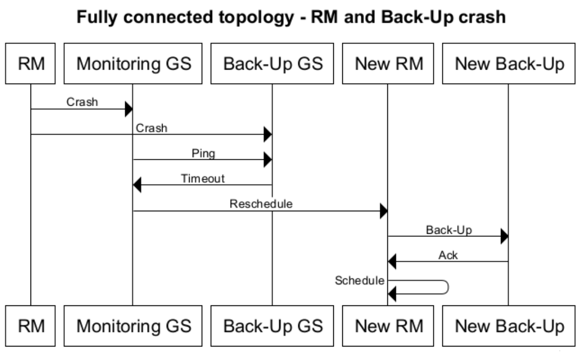

Figure 7: Fully connected topology - RM & Back-up Crash recovery

## B.4 Fully Connected Topology - Monitor & Back-up Crash recovery

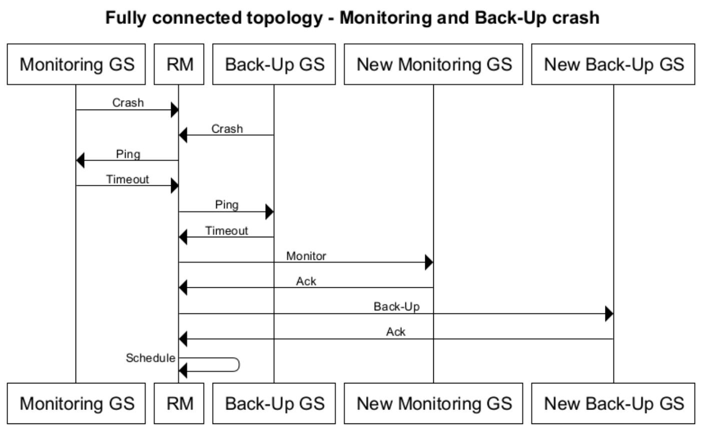

Figure 8: Fully connected topology - Monitor & Back-up Crash recovery

## B.5 Fault recovery

Here is an overview of the logs from recovery of the different fault situations.

### B.5.1 Resource manager crash

```
[RM 0] Received job 0
[GS 0] Monitoring job 0 on rm 0
[RM 0] Job 0 monitored at 0
[GS 1] Backing up job 0
[RM 0] Job 0 backed up at 1
[RM 0] Scheduling job 0
[RM 0] Executing job 0
[RM 0] Offline
[GS 1] Recovering rm crash for backed up job 0 at 0
[GS 1] Monitor (gs 0) still up, release job
[GS 0] Rescheduling job 0
[RM 1] Job 0 monitored at 0
[RM 1] Received work order for job 0
[GS 1] Backing up job 0
[GS 0] Monitoring job 0 on rm 1
[RM 1] Job 0 backed up at 1
[RM 1] Scheduling job 0
[RM 1] Executing job 0
[RM 1] Finished executing job 0
[RM 1] Releasing job 0
[U  0] Result for job 0
```

### B.5.2 Resource manager and monitor crash

```
[RM 0] Received job 0
[GS 0] Monitoring job 0 on rm 0
[RM 0] Job 0 monitored at 0
[GS 1] Backing up job 0
[RM 0] Job 0 backed up at 1
[RM 0] Scheduling job 0
[RM 0] Executing job 0
[GS 0] Offline
[RM 0] Offline
[GS 1] Recovering rm crash for backed up job 0 at 0
[GS 1] Monitor down, promote, reschedule
[GS 1] Use backup for job 0
```

```
[RM 1] Job 0 monitored at 1
[RM 1] Received work order for job 0
[GS 1] Rescheduling job 0 as monitor
[RM 1] Job 0 monitored at 1
[RM 1] Received work order for job 0
[GS 1] Monitoring job 0 on rm 1
[GS 1] Promoting to primary for job 0 for rm 1
[GS 1] Monitoring job 0 on rm 1
[GS 2] Backing up job 0
[RM 1] Job 0 backed up at 2
[RM 1] Scheduling job 0
[RM 1] Executing job 0
[GS 2] Backing up job 0
[RM 1] Job 0 backed up at 2
[RM 1] Scheduling job 0
[RM 1] Executing job 0
[RM 1] Finished executing job 0
[RM 1] Releasing job 0
[U  0] Result for job 0
```

### B.5.3 Monitor crash

```
[RM 0] Received job 0
[GS 0] Monitoring job 0 on rm 0
[RM 0] Job 0 monitored at 0
[GS 1] Backing up job 0
[RM 0] Job 0 backed up at 1
[RM 0] Scheduling job 0
[RM 0] Executing job 0
[GS 0] Offline
[RM 0] Recovering from monitor crash for job 0
[GS 1] Promoting to primary for job 0 for rm 0
[GS 1] Monitoring job 0 on rm 0
[GS 1] Releasing back up for job 0
[RM 0] Job 0 monitored at 1
[GS 2] Backing up job 0
[RM 0] Job 0 backed up at 2
[RM 0] Finished executing job 0
[RM 0] Releasing job 0
[U  0] Result for job 0
```

### B.5.4 Back-up crash

```
[RM 0] Received job 0
[GS 1] Monitoring job 0 on rm 0
[RM 0] Job 0 monitored at 1
[GS 0] Backing up job 0
[RM 0] Job 0 backed up at 0
[RM 0] Scheduling job 0
[RM 0] Executing job 0
[GS 0] Offline
[RM 0] Recovering from back-up crash for job 0
[GS 2] Backing up job 0
[RM 0] Job 0 backed up at 2
[RM 0] Finished executing job 0
[RM 0] Releasing job 0
[U  0] Result for job 0
```

### B.5.5 Monitor and back-up crash

```
[RM 0] Received job 0
[GS 1] Monitoring job 0 on rm 0
```

```
[RM 0] Job 0 monitored at 1
[GS 0] Backing up job 0
[RM 0] Job 0 backed up at 0
[U  0] Accepted job 0
[RM 0] Executing job 0
[GS 0] Offline
[RM 0] Recovering from back-up crash for job 0
[GS 3] Backing up job 0
[RM 0] Job 0 backed up at 3
[GS 1] Offline
[RM 0] Recovering from monitor crash for job 0
[GS 3] Promoting to primary for job 0 for rm 0
[GS 3] Monitoring job 0 on rm 0
[GS 3] Releasing back up for job 0
[RM 0] Job 0 monitored at 3
[GS 2] Backing up job 0
[RM 0] Job 0 backed up at 2
[RM 0] Finished executing job 0
[RM 0] Releasing job 0
[U  0] Result for job 0
```

## Appendix C Attribution

Modern software development would not be the same if not for open source development. To show our gratitude for those who have made their work freely available, we attribute the tools used in the making of this project and report.

**Java** Java was used for the initial implementation.

**ParSeq by LinkedIn** ParSeq is a library for asynchronous programming in Java.

**LogBack** . LogBack was used for logging

**JUnit** JUnit was used for testing in java.

**Scala** Scala was used for the later version of the project.

**Matlab** Matlab was used for the fairness analysis.

**ScalaTest** ScalaTest was the testing framework for the later Scala implementation.

**WebSequenceDiagrams** Used to draw sequence diagrams to clarify the workflows.

**Google Docs** Used to draw the boxplot.

**ShareLatex** This document was edited online using ShareLatex.

## References

[1] C. Pettitt et. al. *ParSeq framework* GitHub Repository, Mountain View, CA, USA, 2012.

[2] Dekker, A. H. and Colbert, B. D.: *Network robustness and graph topology*, in Proceedings of 27th Australasian Computer Science Conference (ACSC2004), Vladimir Estivill, Castro, Ed., ISBN 1-920682-05-8, Dunedin, New Zealand. CRPIT, 26. Estivill, Castro, V., Ed. ACS, pp. 359-368 2004,.