Distributed Mass Image Processing in a Cloud Environment

Christos Froussios
4322754
C.Froussios
@student.tudelft.nl

Richard van Heest
4086570
A.W.M.vanHeest
@student.tudelft.nl

Alexandru Iosup
Course instructor
A.Iosup@tudelft.nl

Dick Epema
Course instructor
D.H.J.Epema@tudelft.nl

Alexey Ilyushkin
Teaching Assistant
A.S.Ilyushkin@tudelft.nl

ABSTRACT

Cloud computing has been increasingly popular over the last decade. It seems promising in some ways, given the successes with cloud based systems of some large companies. However, there are drawbacks on this kind of systems as well. In this report we describe our findings of one month of research in the area of cloud computing. We build a small prototype of a IaaS-based application that does image processing and runs on the Amazon Web Services (AWS) EC2 cloud platform. The main focus points in the design and research of this prototype are automation, scalability, load balancing, reliability and monitoring. To solve issues like scalability and load balancing, we develop several policies to be tested and compared against each other. Furthermore we measured the performance of the system by applying several metrics. Finally we come up with an analysis of the pro's and con's of switching to cloud computing.

1. INTRODUCTION

Cloud computing has gained an increasing interest in the last couple of years. In contrast to the era of grid computing, not only the academic world is interested in this new way of performing large scale computations, but also companies do so. The most logic explanation for this shift of interest is that computer grids are expensive to invest in for a company which may not use the required hardware constantly, whereas in cloud computing that company would lease the required machines, scale up or down if appropriate and only get charged for the hours between leasing and releasing.

We distinguish three types of cloud computing:

Software as a Service (SaaS) applications such as Google Drive, Dropbox, Gmail, Yahoo mail and Facebook.

Platform as a Service (PaaS) the computing platforms which typically include operating systems, programming language execution environments, databases and web servers. Examples of this are Apache Hadoop, Google App Engine, Windows Azure and Amazon Web Services Elastic Beanstalk.

Infrastructure as a Service (IaaS) the computing infrastructure, physical or (more often) virtual machines and other resources like virtual-machine disk image libraries, block and file-based storage, firewalls, load balancers, IP addresses and virtual local area networks. The most prominent products in this category are Windows Azure, Google Compute Engine and Amazon EC2.

In this report we will focus on IaaS cloud computing especially, as requested to investigate by WantCloud BV. We will look into the advantages and disadvantages of IaaS by implementing a small prototype for a potential future system. The main features of this prototype can be summarized as:

Automation working as much as possible independently from any human interaction

Elasticity (auto-scaling) leasing and releasing machines from a resource pool as workloads change over time

Performance (load balancing) allocating workloads to machines from the resource pool in such a way that the machines are used as effective as possible.

Reliability building in a fair amount of fault tolerance

Monitoring observing and recording the current status of the system as well as measuring its components' performance by applying various metrics.

Multi-tenancy having multiple concurrent users in the system while discriminating between their importance.

The prototype to be implemented will be an application that receives pictures from an external source (for example a web form where users can submit their pictures), performs some operations on them and returns the processed pictures. These operations may vary from lightweight operations such as a flip of the picture to heavier operations like Fourier transform.

The application is set up in such a way that all pictures are received at a head node, which allocates them to one of the worker node in its resource pool and leases new machines or releases idle machines when appropriate. We present and compare multiple policies for these problems.

In this report we will focus on the general idea and the headlines of this project. If you are interested in implementation details, it is available at GitHub[1].

This report is structured as follows: first we will discuss the background of the application in Section 2. What does

[1] `https://github.com/rvanheest/IN4392_Cloud_Computing_Labs`

it do in detail? What are its requirements and features? This is followed by the system design in Section 3, where we focus on the internal workings of the application as well as several policies for both the processes of allocation and leasing/releasing nodes. We evaluate the application in Section 4 by applying several metrics. Finally we close the report with a discussion in Section 5 and the conclusion in Section 6. In Appendix A an overview of spent time is provided.

2. BACKGROUND OF THE APPLICATION

The application we build over the course of several weeks is a prototype of a more generic cloud centered solution where tasks come in at the head node and are distributed over a number of worker nodes. In this particular application we choose to take image processing as the workload. Pictures that are received by an arbitrary worker are modified by applying a number of operations, combining them afterwards. Notice that this approach has a high potential for the task being split into subtasks and make it into a MapReduce process. However, to make things not overly complicated in this prototype, we consider one picture as one *undividable* task, which is executed on a single node.

The basic requirements of our application can be split up into 5 tiers. First of all, the application needs to run without human intervention as much as possible. We fulfilled this requirement by only having to start the head node application and providing pictures over the course of the runtime. Besides these 2 acts, the head node automatically decides when to lease and release worker nodes, to which worker an incoming task is sent and to where a completed task needs to be returned. On the other hand, the worker nodes automatically receive tasks, process them as soon as they possible can and make sure they send back the result to the head node.

A second tier holds the elasticity (or scalability) of the system, which is determined by the head node. Elasticity in the context of cloud computing describes to which degree the system is able to grow and shrink its number of resources. In this case, these resources are the worker nodes, which can be leased or released. When the system gets overloaded with tasks, it might be a good idea to lease more workers. On the other hand, when there are hardly any tasks in the system, it is useless to have a large number of idle workers, since they are being payed for all that time. Ideally you want to predict when the system will have a peak moment in the number of tasks and when it will get more quiet. For this, machine learning is a nice path to follow. This however is not part of this project.

Besides elasticity, load balancing is another topic of interest in cloud computing. When the head node receives a task, it needs to decide where it should be executed. Does it prefer certain workers, will it be assigned randomly or is there a better way to schedule a task. In this the head node might look at the size of the task, the length of the queue of each available worker or any other measurable property of the task and the workers.

The fourth tier is the reliability of the system. As the application is mostly autonomous, it also needs to be able to recover itself from failure. A failure might be losing connection between the head node and one or more workers, crashing the head or a worker, IO failure, etc. For this project we make the assumption that the head node never crashes and that it never completely loses access to the network.

Finally, a system isn't a good system without a monitoring module. To make decisions in the other tiers (especially elasticity and load balancing), we need to supply it with proper data, for example the length of each worker's queue, the number of incoming tasks in the system, etc. Also for the research part of this project the monitoring module is used. This is used to gain the results in Section 4.

Besides these 5 tiers, we added support for multi-tenancy. It is possible for multiple users to connect to the head node and send in their images. At this point, the system does not discriminate between the importance of users and uses the policy of "First-come, first-served".

3. SYSTEM DESIGN

The application we present here can be seen as a prototype for a typical cloud centric work flow, where a user sends workloads to a head node, which distributes it over one or more workers (depending on whether the workloads are splittable into subtasks). Once the workload is processed, a result is send back to the head node, which forwards this result to the appropriate user. Based on the workloads, the head node needs to decide to which worker a workload is allocated and whether or not to lease or release workers, for which it uses certain policies.

In this section we will discuss the system's architecture, work flow, communication between user, head node and workers, followed by the policies that are used in scaling and load balancing. Finally we will describe the additional features that is in the system: multi-tenancy. Where possible we will give a more general description of the design, rather than describing the system for the current image processing application. This might help in developing future projects based on the same principle.

3.1 Architectural overview

In order to understand the system's architecture and work flow, let's follow a workload as it works its way through the system. We refer to Fig. 1 and Fig. 2 for an overview of the architecture.

The user who sends this workload will sign in to the head node via a socket connection. This connection request comes in at the `ClientReception`, which creates a `ClientHandle` for workloads to be received. From this point on, the user interacts directly with its own `ClientHandle` in the head node. When this receives a workload, it makes a local copy of the workload and adds it to the `TaskQueue`. This is where all workloads come together while waiting to be processed. It should be noted that (for reasons explained later) the `TaskQueue` is a double-ended queue, meaning that tasks not only can be pushed at the end of the queue but also at the beginning.

The `Monitor` continuously samples the current state of the head node. For this reason, it also knows that the workload is queued. Based on the total workload in the queue (and other metrics and policies that are explained later), it decides whether or not to lease extra workers. When extra workers are needed, the `Monitor` makes a call to the Cloud Service to request a new machine.

The `WorkerReception` waits for this machine to contact the head node. Once the socket connection is established, a `WorkerHandle` is created and stored in the head node, via which all future communication with the newly leased ma-

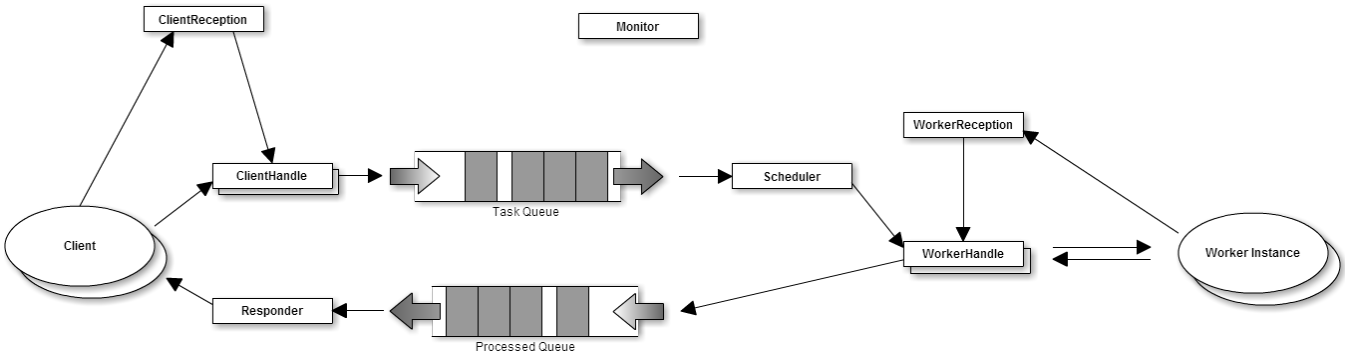

Figure 1: Architecture in the head node

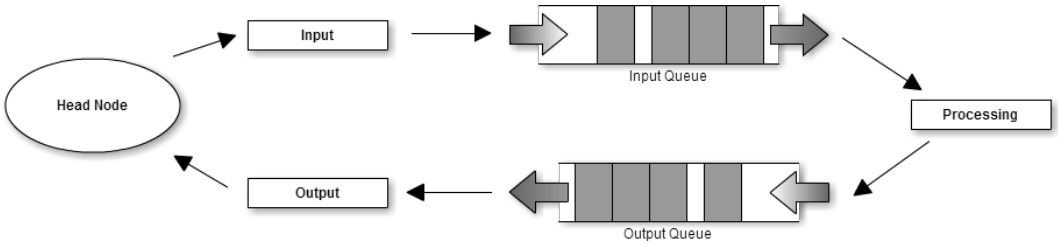

Figure 2: Architecture in a worker

chine will take place. The first communication is the worker who introduces itself and tell how many cores it has. This information is stored in the `WorkerHandle` for later use. Once this formal procedure is over, the new worker will wait for the first workloads to come in.

Besides coordinating the leasing of workers, the `Monitor` is also responsible for cleaning up workers that are not needed anymore. This is done based on metrics and policies that will be discussed later. When it decides to release a worker, the corresponding `WorkerHandle` will be flagged as being set for decommission. After that, no more tasks can be scheduled on that worker; it will only finish its currently scheduled tasks, after which is is released.

The `Scheduler` takes workloads out of the `TaskQueue` and decides on which workers they will be processed. This subsystem takes all tasks out of the queue and collects all eligible workers (the ones that are not flagged for decommission). Based on a predefined policy, an allocation plan is made where tasks are assigned to certain workers. It is possible in some policies that not all tasks can be scheduled at once. For that reason, not only the allocation plan is returned by the policies, but also a list of not yet scheduled tasks. The latter will be put back on top of the queue, so that they are the first to be scheduled again next time. The tasks that were in the allocation plan will get assigned to the appropriate workers by sending their workloads over the previously established socket connection. In the case the connection failed in the meantime, the task is put back in the queue as well to be rescheduled.

When a task is allocated to a certain worker, its workload will be send over there via the socket connection. At the worker side, it is received by the `Input` subsystem, which puts it in the `InputQueue` for it to be processed. When the `Processing` subsystem is finished with the previous task, it will take the workload out of the queue and performs the actual processing. After that is finished, the result is put into the `OutputQueue`, from where it is send back to the head node by the `Output` subsystem.

At the head node, the result is received over the socket by the `WorkerHandle` which removes the corresponding task from the list of currently processed jobs and adds the result to the `ProcessedQueue`. There it waits until the `Responder` takes it out and sends it back to the user.

3.2 Architectural future improvements

The architecture described above is what the application currently looks like. However, this does not fully satisfy the redundancy requirement. This is due to this project's deadline. In the following we describe what to change in and add to the architecture to fully support this requirement.

The head node is the system's single point of failure. To defend against failures in the head node, the node can be mirrored. Despite being significantly lighter than worker nodes, the head node can also be overwhelmed in a busy real environment. To improve scalability, it is possible to split the head's components into a set of closely collaborating nodes. In that case, only the scheduler and monitor need to be mirrored. Other components can naturally collaborate with duplicates of themselves.

When the connection to a worker fails, the head node removes that worker from the worker pool and continues serving requests using the rest of the workers. All the tasks assigned to that worker are presumed lost. If the failure was temporary, the worker is responsible for re-initiating connection to the head node and re-entering the worker pool as a new worker. In the current state of our implementation, workers do not attempt to recover. This was not deemed necessary, as such failures are so rare (we did not encounter any) that we did not consider them a significant resource leak. In a more robust implementation, a worker should

attempt to publish its log files and stop, so as not to sit idle.

As we already said, tasks assigned to a failed worker are presumed lost, since one cannot reliably expect the worker to recover on time. That leaves us with the problem of resuming work on a different worker, in the case that the head node does not have enough RAM memory to backup every request, or enough storage IO bandwidth to store them on a local drive. For that, we would propose a third node role: the file node. That node serves as a remote hash map, that stores tasks given to it and serves them when requested with their unique id. The head node, upon receiving a request, passes it on to one or more file nodes to be stored. When scheduling the task, rather than transmitting the workload, only the identifier and a list of file nodes is sent. The worker then retrieves the workload from the closest available worker. The result of the task is then returned to the file nodes and the head is signalled to retrieve it and send the response.

Since the file node is an IO-heavy process, while the worker node is CPU-heavy, it is recommended that both are deployed on each machine to maximize the use of the machine's resources. That has the additional advantage that if the file node that holds a task and the worker that has to process it are on the same machine, loading the image is done locally and network resources are preserved. A special task scheduler can be designed so that it shows preference to the worker that resides next to a file node that holds the task.

3.3 System policies

When it comes to scheduling tasks or leasing and releasing workers, the approach can influence the performance of the whole system. In this section we discuss the different policies we created for both scheduling, leasing and releasing.

Scheduling policies

Every scheduling policy implementation receives a list of tasks to be scheduled as well as a collection of eligible workers and has to return both an allocation plan and a list of unscheduled tasks. We first present some simple policies, followed by some more sophisticated policies that are all scheduling all provided tasks. After that, we modify the latter policies to only schedule some of the tasks while rejecting others.

A very basic scheduling policy, the `RoundRobinScheduler`, does not need any knowledge about the workers or the tasks. It allocates the tasks by iterating over the collection of workers. It treads every worker and every task in an equal way, giving no priority to any of the tasks and not regarding the length of the queue of a certain worker.

A modification of this policy would be to not iterate over the collection of workers but instead allocate the tasks randomly to certain workers, hence the name `RandomScheduler`. This might cause an unbalance in the distribution of tasks on a single scheduler run, but given simple statistical methods and theorems it will ultimately give a more balanced solution in the long run.

The previous policies might be used when the scheduler for particular reasons is not allowed or able to know the current state of the workers or the contents of the tasks. However, when this restriction is not there, an obvious more sophisticated policy is the `QueueLengthScheduler`, which makes the allocation decision based on the number of tasks that are already present in that worker. The worker with the least amount of tasks will be the one to execute the task at hand. This however asks either for a lot of communication with each of the workers to check their status or a good bookkeeping solution in the head node. The decision for this trade-off can be based on the amount of computer memory versus the speed of the communication. Also this policy assumes that all tasks are of equal processing length, which is more than often not the case.

A '*future work*' policy that takes this processing length into account is the `QueuePixelScheduler`. This is a somewhat special policy for our case of image processing that follows from the way our image processing is implemented and can be summarized by the conclusion that the number of pixels in a picture is proportional to the processing time of that picture. Therefore this policy modifies the `QueueLengthScheduler` by taking into account *the number of pixels* rather than *the number of tasks* in a particular worker. Although this is a direct application of the image processing case, the principles of this can be used in other areas.

Until this point all policies have allocated all the provided tasks to eligible workers. We defined the eligibility of a worker as not being flagged for decommission. However, when we extend this definition by following the philosophy that a worker should only have $2n$ tasks in its system (n tasks in process and n tasks in its queue, where n is the number of processors in that worker node), we end up with a smaller set of workers that are available to receive new tasks. This means that the tasks stay in the head node as long as possible, rather than waiting in a queue in the worker node, which has the advantage that the worker can focus more of its time on processing the tasks rather then being busy with receiving tasks in its queue. Therefore the internal memory of the worker nodes can be used more for the execution of the tasks rather than storing dozens of next jobs. Besides that, using this type of policies makes the system easier to recover from a failing worker, since there are only two tasks in that worker that need to be rescheduled. We applied this paradigm to the `QueueLengthScheduler` and used this in the experiments presented in Section 4. In '*future work*' the `QueuePixelScheduler` will also be adapted to this paradigm.

Leasing and releasing policies

To decide on scaling the system up or down, the state of the system is sampled in given intervals. The information gathered includes the amount of tasks in the queue, in each worker, the total number of cores available and the number of workers that have been requested but have not connected yet. With this information we construct a workload metric W, which, as seen in one of our experiments in Fig. 3, predicts the scheduling delay:

$$W = tasks/cores$$

where *tasks* is the set of all the tasks in the queue and the workers, and *cores* are the total number of available cores in the workers. Generally, when $W > 1$, the system has been receiving jobs faster than it can process them, while $0 < W < 1$ means that the system has enough resources.

We use W to decide on leasing or releasing more workers. Because one worker takes a significant amount of time to boot and W can be expected to stay above 1 while the resources are under way, we introduce the metric of *promised*

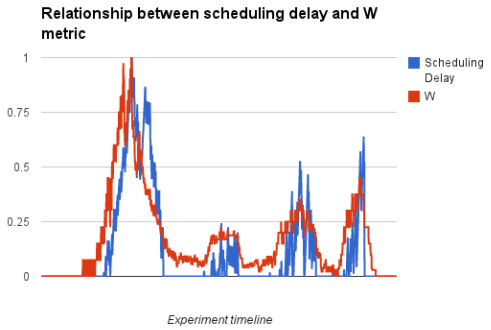

Figure 3: The metric W correlates with scheduling delay

workload which also considers booting machines:

$$PW = tasks/(cores + cores_{promised})$$

With this metric we avoid the situation that the system will repeat resource requests, because machines are not ready yet.

The last optimization we attempted was to lessen our system's responsiveness to demand spikes. Since workers take a while to boot, it is possible that the spike is already served by the time the workers become available and thus the resources are not put to use. For that reason we reduce the responsiveness of the system using an exponential smoothing function

$$SPW_{i+1} = s * SPW_i + (1 - s) * PW_{i+1}$$

where $0 \leq s \leq 1$ is the smoothing factor. We've set $s = 0.95$, with the sampling rate being $1Hz$.

We base our decisions for leasing and releasing machines on metric SPW. More specifically, we aim to keep it in $0.5 \leq SPW \leq 0.8$. When SPW is below 0.5 for a given amount of time, a machine is released. When SPW is above 0.8, a request for a 25% increase is immediately performed. Note that our policy for scaling up is more aggressive than our policy for scaling down. There are two reasons for that decision. The first reason for that is that we choose to err on the side of having more resources than necessary. The second reason is that leasing resources is a slower operation. Once the workload is exceeds the system's capacity, jobs accumulate in the queue until more workers arrive. Even more resources will be needed to consume the queue and return the scheduling delays to acceptable levels. For that reason we also allow the system to issue multiple resource requests, even if the first batch of workers is still booting. The system will scale down again once the queue has been consumed.

3.4 Additional System Features

The additional feature covered in this prototype that was not required in the set of basic requirements is multi-tenancy. This term is defined as an IT sharing model of how physical and virtual resources are used by possibly concurrent tenants. In our application this means that multiple users can concurrently use the system, all connect to the head node, send in their workloads and all receive back their own result without seeing any results of other users.

In the research field of multi-tenancy, this brings up the philosophical question of fairness. Is it fair to discriminate between users that are very active and send in workloads more regularly than others or should a less active user have a higher priority? In this prototype we decided not to get into this discussion, just treating every user equal, using the policy of First-Come, First-Serve.

4. EXPERIMENTS

4.1 Experimental setup

Although the internal structures might differ, the general workings of all IaaS cloud services are mostly the same. Whether you choose for Amazon EC2, Microsoft Azure or another, the approach taken in the previous sections would be the same on architectural level. For this prototype we decided to use Amazon EC2, mainly because both authors already had good experience with it.

In order to lower the expenses on this project, we made use of the EC2 free tier machines. Each of our machines was an instance of a preconfigured Ubuntu t2.micro image. Computing resources for these virtual machines is done with a credit system that allows one to exceed its nominal performance, but suffer limits later on to make up for the burst. This was a significant problem for us, as the performance would plummet to less than 10% of the burst. As an example, an image that is processed in 3 seconds in a burst, would take 40 to 60 seconds. To make things even more complicated, the limits are not applied across all workers, which made worker performance very inconsistent. Our scheduling and scaling policies remained valid under the circumstances, but there is significant noise in our experiments nonetheless. We could not reduce the noise by averaging multiple experiments either, since the cause follows a strict pattern, starting with high performance and dropping shortly after. Another limitation to our experiments what that free-tier machines are limited to 20 at any given moment. This limited the scale of our experiments to use cases that could be served by 20 machines with low performance.

In order to send tasks to the system for the purpose of experiments, we built a bot that sends tasks to the head node. With this we are able to send continuous streams of tasks (with user-defined intervals between requests) using a command line, as well as multi-tenant predefined experiments.

We will present here two cases. In the first experiment, demand rises and falls repeatedly but gradually over the course of a few minutes. Demand peaks at 1 image per second and the whole experiment lasts approximately 25 minutes. In the second experiment, our system is surprised by a steady stream that exceeds its initial capacity. More specifically, the system starts with two workers available and the stream is one image every 2.5 seconds for 10 minutes. With the processing times given earlier, we estimated that, with the worst performance by the workers, the system would need to scale 15 workers.

4.2 Experimental results

In the first experiment, we emulated users entering and staying in the system for a few minutes. We modelled that so that peaks and valleys of demand are created. The system was expected to scale up and down accordingly. In Fig. 4 we can see how the system adjusted itself to the demand, thus demonstrating its elasticity. Demand is measured by the

metric W, which we already showed to capture the system's need for more resources. We do not show the frequency or requests, as it doesn't correctly predict the need for resources, due to the workers' inconsistent performance. Apart from the first peek, where the system was overwhelmed, the system adjusted to demand smoothly. We attribute that first peak to the moment that our first workers' performance was capped, cutting our system's throughput by 90%. In Fig. 5 we can see how much time each task had to spend in the system, outside of being processed. We call this time *overhead* and lower values are better. As can be seen, the vast majority of tasks experienced little to no overhead.

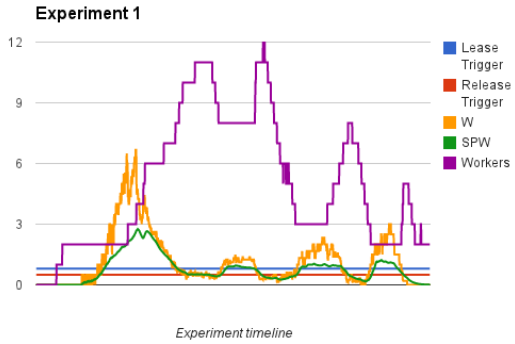

Figure 4: Monitored information about the system in the first experiment

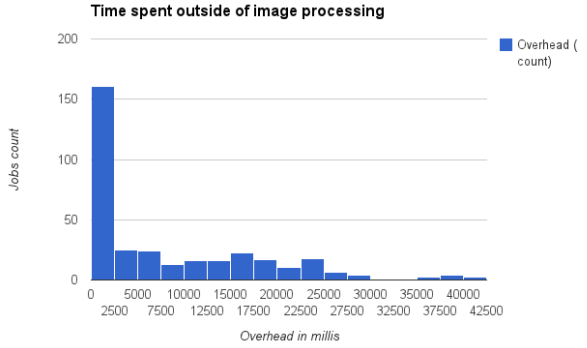

Figure 5: Overhead time for jobs in the first experiment

In the second experiment, we wanted to test how the system would behave when overwhelmed by demand. As can be seen in Fig. 6, the system was overwhelmed at first, but succeeded in catching up to the demand and consumed the accumulated work. In Fig. 7 we can see that the system performed much worse in this case, as expected. The overhead reached up to 2 minutes, due to the tasks waiting in the queue. For scale, the whole experiment lasted 10 minutes.

We identify leasing times as the biggest cause of poor performance in our system. While demand scales up, scaling the system happens with a delay of 1 to 2 minutes. Requests issued in the meantime experience delays.

We attempted to run the first experiment with a more generous leasing policy, that aims to keep $0.3 \leq SPW \leq 0.5$. To our surprise, the results were significantly worse, as seen in Fig. 8. We do not consider that to be the result of the policy. Instead, this shows us that our IaaS provider's throttling policy was the most influential factor in our experiments. For that reason, we consider running further experiments in these conditions a futile process.

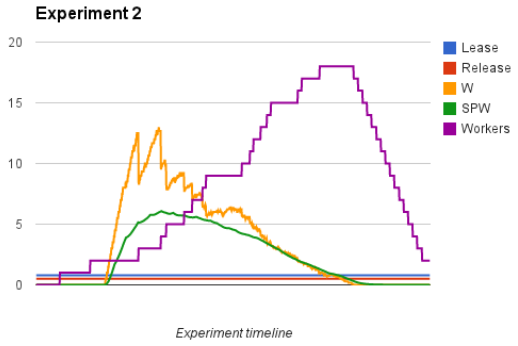

Figure 6: Monitored information about the system in the second experiment

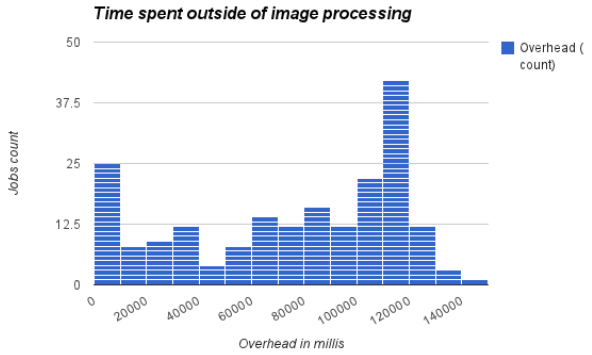

Figure 7: Overhead time for jobs in the first experiment

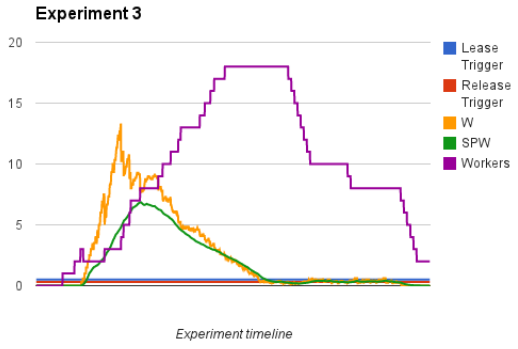

Figure 8: Monitored information about the system in the third experiment

5. DISCUSSION

In the past month, we built a prototype of an application based on IaaS clouds. At first glance, the architecture is very straight forward and easy to extend. However, under the hood it can become quite difficult, especially when it comes to the policies part of the system. What is a good policy for scheduling, leasing and releasing? What is the most optimal combination of them all? How do we define fairness regarding the multi-tenancy? How do we make the system fault-tolerant and not do redundant processes or slow the system down performance wise? Trade off analyses have to be made for those topics.

Also, in our current implementation, we have tried to stick to the standard Java library as much as possible. The only external dependency is the Amazon EC2 API. However, the standard Java library has some drawbacks regarding multithreading and data structures. Therefore we would advise to continue the development by writing code in a more functional style, where thread-safety and concurrency is easier to manage, safer to execute and less error prone. Also we recommend switching to a push-based model like ReactiveX[2], which gives an extra boost in performance, as is shown by companies like SoundCloud and NetFlix[3], rather than the pull-based model where you are continuously blocking threads while waiting for activities to happen.

Cloud computing has a number of benefits. First of all, it enables the company to focus more on the business, rather than on managing data centers. Instead of having an entire infrastructure in the company, it is financially way better to rent the infrastructure from companies that are specialized in building, managing and maintaining cloud facilities. Especially the Pay As You Go model, where you only pay for the hours you use, is very attractive for systems with fluctuating resource needs. Also the fact that you don't need to invest in replacing aged hardware is a large benefit. Besides that, cloud computing will allow you to expand your global presence.

On the other hand, there are a number of downsides to cloud computing. Cheaper virtual machines are typically not dedicates, and thus can experience a lot of interference. Because of that, the system's performance will be inconsistent. Besides that, cloud computing may not be the best fit for your type of workloads. Some may have very specific performance and security requirements, which are not always guaranteed in a cloud environment. Finally we have to point out the general user community is not always convinced about the security of the cloud. However, a cloud product is as safe and secure as you make it. Using good firewalls, network gateways, anti virus and compliance scanning will help with the security issues.

Another downside of using the cloud is that leasing new machines sometimes takes very long, mostly in the order of minutes (this actually depends on the cloud provider, as well as the type of machine requested). The problem with that is that, for workloads that fluctuate in a matter of minutes, often the workload for which the extra worker is leased is already consumed or the workloads have piled up while waiting for the additional machine. We see two separate solutions for this issue. One is to optimise for the lease time of machines. Rather than performing a new lease and a cold start when a new machine is needed, a buffer of pre-leased inactive can be maintained. A free-tier machine on an SSD drive in Amazon's EC2 takes half the time to do a warm start (from the stopped state) than it needs to do a cold start. The drawback of this is that you have to pay for the machine's storage, even when the machine is stopped but still in the pool of available machines. This isn't the best warm start possible either, since some resources (e.g IP address) need to be reassigned and a complete boot is performed. IaaS providers may provide even faster alternatives, though we did not have any available to us.

The second solution is to improve the metric for workload prediction. Although this is not easy, since it asks for sophisticated and expensive to research methods like machine learning and feedback loops, it might be the best and cheapest way to go in the long run.

In our experiments, we experienced poor performance when demand rose. One reason for that was the leasing times. Another reason was that, due to the small scale of our experiments, small changes in demand where big proportionally to the system. We expect that in bigger environments, the system would scale with greater success. Since workers are independent from each other, or, given our suggestions for redundancy, depend on a constant number of other nodes, we believe that the only bottleneck for scalability will be the head node. The head node is quite light compared to a worker and, given a powerful machine, should be able to manage thousands of workers. Going beyond that, the head also has a finite potential for becoming distributed, as mentioned in 3.2, which can increase scalability by an order of magnitude. For even greater demand, it is possible to duplicate the system and redirect requests so as to distribute them. Those systems do not need to collaborate at all and their scalability is infinite and limited only by the scalability of the DNS in front of the system.

6. CONCLUSION

In this report we discussed our findings and results from one month of research in the area of cloud computing. We build an IaaS cloud based application that processed images. Furthermore we described the general architecture of such an application, containing task queues, scheduling, lease and release policies, system monitoring, fault tolerance, multitenancy, as well as communication strategies.

Cloud systems take up a lot of time to develop, especially since it is still a new technology and there are many combinations of policies to choose from in order to optimize the performance. Therefore we described throughout the report subjects that are not in the application yet, but that will be added later. For now they were there to show what is possible and in which direction to go.

Although it is still a new technology and cloud applications are harder to develop and maintain, it is definitely an area to look into for future expansion. Cloud computing is very promising, has a lot of potential and is interesting on the financial side as well. If the demand to be met requires a distributed system, then leasing hardware from a IaaS provider will have a lower entry and maintenance cost, and the service we implemented is definitely scalable on a cloud environment. Therefore we would conclude by advising to invest more in this area, both knowledge wise and financially.

[2] `http://reactivex.io/`

[3] `http://techblog.netflix.com/2013/02/rxjava-netflix-api.html`

APPENDIX

A. TIME SHEET

This appendix holds an overview of the number of hours spend on this project, divided in several categories. For a more detailed overview, we refer to our GitHub repository[4].

Category	Time spend (hours)
Think	24
Develop	103
Experiment	12
Analysis	3
Write	37
Waste	37
Total	216

[4] https://github.com/rvanheest/IN4392_Cloud_Computing_Labs