CloudSicle: A Cloud Implementation of Gifsicle

Christiaan Titos Bolívar (C.TitosBolivar@student.tudelft.nl), Quinten Stokkink (Q.A.Stokkink@student.tudelft.nl)

Course Instructors : Alexandru Iosup and Dick Epema PDS Group, EEMCS ([A.Iosup,D.H.J.Epema]@tudelft.nl)
Lab Assistant : Bogdan Ghit PDS Group, EEMCS (B.I.Ghit@tudelft.nl)

***Abstract*—This report shows CloudSicle.An IaaS-based implementation of Gifsicle, for the creation of animated *.gif* images. We show the modular framework that makes up CloudSicle and some possible strategies that could be made for it. The results of two types of experiments are evaluated and it is found that the current proof of concept for CloudSicle is lacking a good allocation strategy. For the current strengths of CloudSicle it is also recommended WantCloud BV reconsiders its use of Cloud-based processing for generating *.gif* images.**

I. Introduction

SINCE the dawn of the internet people have sought to enrich their internet experience. As we have gone from plain text to richt text to eventually images accompanying text, in 1987 we eventually ended up with the Graphics Interchange Format or *.gif* which made its way onto our webpages.

We have seen databases work their way onto the internet to communicate data to users (e.g. SQL) and we have seen Cloud storage systems to communicate files over the internet (e.g. Facebook, Google Drive, etc.). In other words, we have seen these developments which are the respective counterparts of delivering textual content on a large scale and delivering images on a large scale. Following the history of the internet, the obvious next step would then be delivering *.gif*s on a large scale. This is where CloudSicle comes into play.

CloudSicle is an application we developed as a proof of concept for **WantCloud BV**, to check the feasibility of Cloud Computing to deliver a rich internet experience. The goal of the application is to process multiple images into one (animated) image in a large scale context. For the sake of implementation we have focused ourselves on the communication between a client, a server and allocated Virtual Machines instead of trying to integrate this all into a webpage context. Furthermore, as our scope lies with the scalability and speed of the Virtual Machine processing, we have outsourced the creation of the actual *.gif*s to an application called *Gifsicle*[1]. *Gifsicle* is a free and open source command-line tool created by Eddie Kohler for the creation of animated *.gif*s.

In the end we hope to present **WantCloud BV** with an implementation design and concept, which can be used to provide a *.gif* creation service to other businesses. The need of the future.

C.Titos Bolívar and Q. Stokkink are students of the Computer Science department of the Delft University of Technology (TUDelft)

The structure of this report will be as follows:

- In Section II we will lay out our requirements.
- Section III will explain the design of CloudSicle as well as some implementation details.
- We evaluate CloudSicle with a number of experiments shown in Section IV
- In Section V we will discuss the main findings of our work as well as the trade-offs of using CloudSicle.

II. Requirements

In this section we will first provide some background information concerning CloudSicle. Following this we will list the requirements we have formulated together with **WantCloud BV**.

Animated *.gif*s are a popular internet phenomenon. These images usually are short in duration. Long animated *.gif*s result in a large file size and thus a large loading time online. The duration of an animated *.gif* often is no more than a 2 to 6 seconds. At a framerate of 24 fps we assume the average number of frames of an animated *.gif* is about a 100. We therefore choose this as the average workload per request CloudSicle has to work with.

As mentioned in the previous section we designed CloudSicle for future deployment in a large setting. This means that CloudSicle should scale very well according to usage. Apart from scaling up we also define coping with varying levels of usage as scaling down in a timely fashion as well. Scaling down can improve resource utilization and can reduce costs as well as we only pay for what we use.

Our second target was to have the system respond as fast as possible. Creating animated *.gif*s is fast, and thus the overhead of the cloud infrastructure should be kept to minimal. This means we would like to always serve requests as they come in.

The summarization of our requirements is therefore as follows:

Automation The system should handle the resource management autonomously.

Intelligent Scaling The system should scale up under high pressure, and scale down under low pressure

Responsiveness The response time and makespan should be kept minimal.

Job Allocation Optimal job partitioning over VMs to ensure that VMs are not inactive during their entire lifespan.

Reliability Failed jobs should be automatically restarted.

Monitoring Administrators should be able to monitor the entire CloudSicle system.

III. System Design

. In this section we will discuss the system design and interactions between these actors and how they apply to the homonym features as requested by **WantCloud BV** (*Automation*, *Elasticity*, *Performance*, *Reliability*, *Monitoring*). The visualization of this behavior is depicted in Figure 1 for reference.

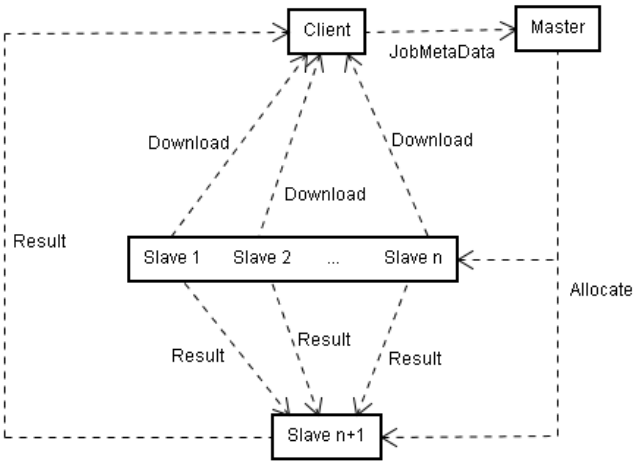

Fig. 1. CloudSicle overall system design for some arbitrary Single Layer VM *SplitJob* strategy

A. Architecture

As previously mentioned the CloudSicle software consists of one or more clients, all of which connect to a server which can allocate Virtual Machines. We will reference the server as the *Master* and the Virtual Machines that it creates as *Slaves*, *Slave VMs* or just *VMs*. The Client, Master and Slaves communicate through structured messages. The application flow is started by a Client sending a request to the Master. This request contains a list of files (as well as data such as the number of files and the total file size) that the client wants to have combined. The Master maintains a job queue, in which all incoming requests are stored before they are scheduled. The Scheduler will select the request at the head of the queue and will then allocate one or more VMs from the Resource Pool to process this request according to some allocation policy. Once the Slave is done, it will send the resulting file back to the Client.

We define several types of messages that can be passed between Client, Master and Slaves.

JobMetaData The request made by the Client is packed in a JobMetaData message. These objects are used by the master to allocate VMs.

Activity An Activity consists of a number of *Jobs* a Slave has to execute in a certain sequence.

StatusUpdate A Slave will send updates about its status to the Master via StatusUpdate messages throughout the execution. For monitoring purposes this also includes the type of *Job* currently being executed.

As mentioned, an Activity message will consist of a list of jobs that the Slave must execute. These possible jobs are:

1) DownloadJob: The DownloadJob instructs the Slave to actively contact some other Slave or the Client and download a given set of files. The Master will set up this situation beforehand, such that the offering party has the files offered for download when the Slave attempts to download them.

2) WaitForResultsJob: The second way to receive input files is using a passive mechanism with a WaitForResultsJob, where the Slave waits for all the input files to be sent over by other Slaves before undertaking action. This is also the only time a Slave will every have to wait before executing its jobs.

3) CombineJob: The CombineJob will process the downloaded input files with *Gifsicle* and stream the output to the Slave's disk.

4) CompressJob: The result of the CombineJob might be further processed by compressing it into a so-called *Tarball*. This is done before the result is sent back to the Client to minimize the network pressure of sending data on the outbound line of the data center. The Master can issue this compression using the CompressJob.

5) ForwardJob: The last step for every Slave VM will be to forward its result, which may be output in the form of a *.gif* file or a *.tar.gz* file (depending on the Activity), and send it to the next Slave or a Client. For the Slave itself it does not matter who will download its result. After performing the delivery to some other party all of the input files and result(s) are removed from its file system and the Slave will either *(a)* signal the Master that it is done with its Activity, marking it as available for the next Activity or *(b)* shut itself down if it has been marked by the Master to exit on Acitivy completion (*SoftExit*).

If any of the Jobs of an Activity fail, the Master will shutdown the VM and reschedule the entire JobMetaData request by putting it at the end of the job queue. This ensures system reliability.

B. Resource Management

One of the most important requirements is the scaling of the resource pool of VMs. If there are few requests, we only need a single VM, if there are many, then perhaps we need more.

The Master maintains a Resource Pool consisting of Slave VMs. The pool has three lists; an *AvailableVMs* list, containing all the VMs that are booted and are waiting for a Job, a *VMsInUse* list containing all the VMs that are currently executing an Activity, and an *allVMs* list containing all the VMs that have been used throughout the lifetime of the Master (this list is for monitoring purposes). Each Slave VM can execute one Activity at a time.

We already mentioned the short nature of the jobs that CloudSicle requests have. When requests are sparse, the overhead of creating a new VM every time would be too large. We therefore make sure we always have one running VM instance in our resource pool. If a request arrives while all VMs are busy, we create a new VM. The scheduler will then wait until a VM becomes available. This could be a busy VM that has

finished in the meantime, or the newly created VM that has finished booting. When a VM is done with its Activity it will added back to *AvailableVMs* list.

To ensure scaling down, after a VM is done with an Activity, the Master will wait a certain amount of time before shutting down the VM and removing it from the Resource Pool. Note that this only happens when a VM is done with an Activity. If it has yet to execute something, it will remain active. Because we always have at least one VM in the pool, and jobs are short, making the need for extra VMs low, the timeout can be very short. In our proof of concept implementation, we set this time out to 6 seconds. In our experiments we show that this value yields good results.

C. Job Allocation

Requests are scheduled in a first come first serve manner. We have defined two allocation schemes for CloudSicle, *SingleJob* and *SplitJob*.

1) SingleJob: SingleJob is a straightforward allocation policy that will simply use one Slave to process the entire request. The Master will pass all the files of the JobMetaData to the DownloadJob of the Slave.

2) SplitJob: SplitJob is a more sophisticated policy. Because it is possible to merge two animated *.gif*s, we are able to divide the input files over multiple Slaves, produce partial results that will then be combined again by another Slave, resulting in an animated *.gif* consisting of all the input files (this is the example shown in Figure 1).

The ForwardJob of the 'intermediate' Slaves will pass the result to the Slave that is designated by the Master to produce the final results. This final Slave will use the WaitForResults job to wait until all other Slaves have sent their partial results. It will then combine and compress as usual and forward the final result to the Client.

This approach bears similarity to a MapReduce computation [2]. Depending on the strategy an arbitrary amount of distinct layers can be introduced. This means CloudSicle can simulate a MapReduce computation with two distinct mapping layers for a given constant amount of iterations (N.B. constant, as the forwarding between Slaves is defined beforehand).

Another strategy that could be simulated is an inverse tree computation for additional speed at the cost of higher overhead (*log(n)* Virtual Machines instead of the constant amount of Virtual Machines required by MapReduce).

D. System Policies

1) Overhead reduction: To avoid Slaves wasting time polling during a WaitForResultsJob, we can have Slaves invoke other Slaves to download files from them. This is done in an effort to shorten the time a Slave waits for its input files and to remove the CPU overhead of busy waiting.

To avoid the Master polling the Slaves for status updates we make the Slaves report their state changes to the Master. To further save on communication cost we can also mark Slaves to exit once their current set of jobs is finished. We can afford this autonomy of Slaves as the jobs they execute are stateless in respect to the rest of the system: if any of the jobs fail, the Slave fails as a whole and if all jobs succeed, the Slave presents only the end result.

2) Security: The security policies for Cloudsicle are limited, but not completely ignored. Communication between the Client and the Master happens through SSH (Secure Shell). This offers some protection from external intervention and limits the set of attackers to individuals authorized on the TUDelft VPN (Virtual Private Network). Any communication between the Master and Slaves, as well as Slaves between each other is handled inside the datacenter (using local IPs). Slaves sending their result back to the Client is done on a default Java SSL (Secure Socket Layer) connection, but still with use of the VPN. In the course of data mitigating over the CloudSicle system this data is also never evaluated, only transformed. As an added security measure all files are only passed on by their file id handle (integer), with only output having a special file id handle. This is a security weak point and might allow external attackers to identify and alter data being sent to Clients.

E. Monitoring

The Master has a Monitor service that keeps track of all the statuses of Slaves in the system (and logs them). It stores JobMetaData for these Slaves and records changes in the Jobs that the Slaves are currently executing (executing *DownloadJob*, executing *CombineJob*, etc.). For Slaves it is recorded whether in what state they are, and the corresponding JobMetaData is then stored in one of the following states:

Waiting The JobMetaData is awaiting allocation to a Slave

Running The jobs of the JobMetaData are being executed on a Slave

Finished All of the jobs of the JobMetaData have been successfully run.

Failed One of the jobs of the JobMetaData has failed.

When the Monitor gets updated by the Master if a change in one of the Jobs happens, it records the time of the event. This allows the Monitor to calculate the average completion time of a set of jobs and the average amount of time spent on each of the jobs. We used the Monitor to get our experimental results (as will be presented in Section IV).

F. Implementation

We have implemented a proof of concept of our design. It is written in Java, and can be found on Github[1].

The CloudSicle project distributes three runnable *.jar* files. These are *client.jar*, *master.jar* and *slave.jar*. It also requires a user to specify his log-in credentials in a *config.txt*, here communication ports for normal messages and file transfers can also be specified.

A client can run his software by placing his *client.jar* and *config.txt* in the same folder and then run the *client.jar*. The client will be presented by a small interface that allows him to select files, the DAS-4 server entry point and send his (automaticly inferred) JobMetaData to the master. When the

[1]https://github.com/ChrisTitos/CloudSicle

activity has been processed the resulting *.tar.gz* compressed folder will appear in the client's folder.

The master will be deployed by sending the *master.jar*, the *slave.jar* and the *config.txt* to the DAS-4 server. The service is then started by running the *master.jar*. The *slave.jar* is automaticly deployed to any Slave VMs the master creates. To stop the service the administrator can simply press the *enter*-key to clean up all allocated Virtual Machines. Should the application crash or be forefully terminated, the administrator will have to shut down all the allocated Virtual Machines by hand.

IV. Experimental Results

In this section we will lay out the setup of our tests our rationale for testing things in this fashion and the results of each test. Apart from the default metrics we will also report the amount of files per hour increments and the cost if the Amazon EC2 rate of 10 Eurocents per hour were used per file.

A. Experimental setup

Our system was deployed on the DAS-4 cluster[3]. For Virtual Machine monitoring we made use of the Java OpenNebula Cloud API [4]. Virtual Machines are created using the `centos-smallnet-qcow2` image available on DAS-4.

To avoid having our tools interfering with the process we measured the makespans (in seconds), for differing input sizes, on the Master (using the Monitor, as discussed earlier) only using the standard Java system calls for timing. These different input sizes consisted of instances of multiple of the same file being fed into the system, each exactly 54870 bytes in size. Due to complications in the implemenation phase, we were only able to test the *SingleJob* strategy.

B. Experiment 1: Makespans

In the first experiment we measured the makespans of different activity sizes. We created datasets of different sizes with the same file. The sizes we created are:

Small A set containing only 2 images
Medium A set containing 20 images
Large A set containing 100 images, which is our expected activity size
Jumbo A set containing 1000 images
Immense A set containing 6739 images, which is the maximum amount *cp* would produce on our system

Figure 2 depicts the running time in seconds for each of these datasets. For the medium and jumbo datasets we have also given a more detailed insight into the specific jobs that are executed in Figure 3.

An interesting observation to be made from Figure 2, is that the time it takes per file to handle a job decreases from 6.5 seconds per file to 0.15 seconds per file (see Table I). If we make a rough inverse exponential approximation of this data we see that the algorithm becomes 'efficient' (second order derivative is 0) at about 500 input files.

We can observe from Figure 3 is that smaller jobs are prodominantly defined by the time it takes to forward their result, accounting for about 50% of the job. As the job gets bigger, we see that this shifts to the downloading taking up most of the time of the job, taking again about 50% of the job. What we can also observe is that compression time takes up a larger percentage of the job time as it gets bigger, waiting time and combination time stay about the same and forward time shrinks percentually.

We expected the forward time and waiting time to be more or less constant with regard to the input size. We see in Figure 3 however, that the waiting time grows with the input size. Upon further investigation it appeared that this is caused due to the increased time the Client needs to send his JobMetaData. The time JobMetaData spends in the waiting queue is virtually the same.

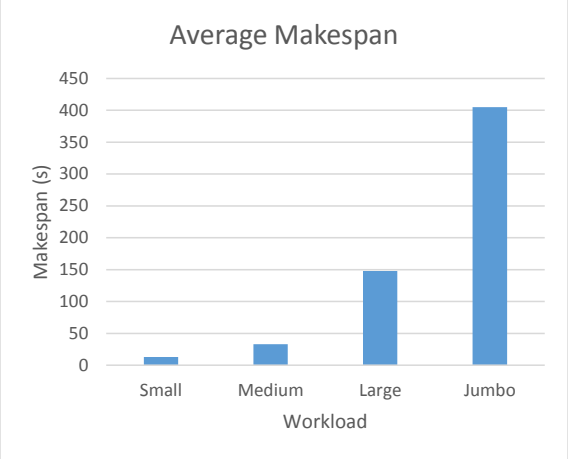

Fig. 2. The average makespan of a single job

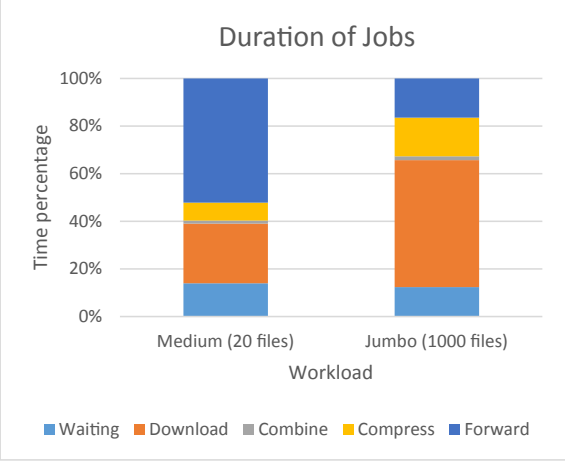

Fig. 3. The makespan of the medium and jumbo workload, average time spent per job type for the activity

C. Experiment 2: Scaling

One of the most important requirements of CloudSicle is the autoscaling of VM resources. To evaluate the autoscaling

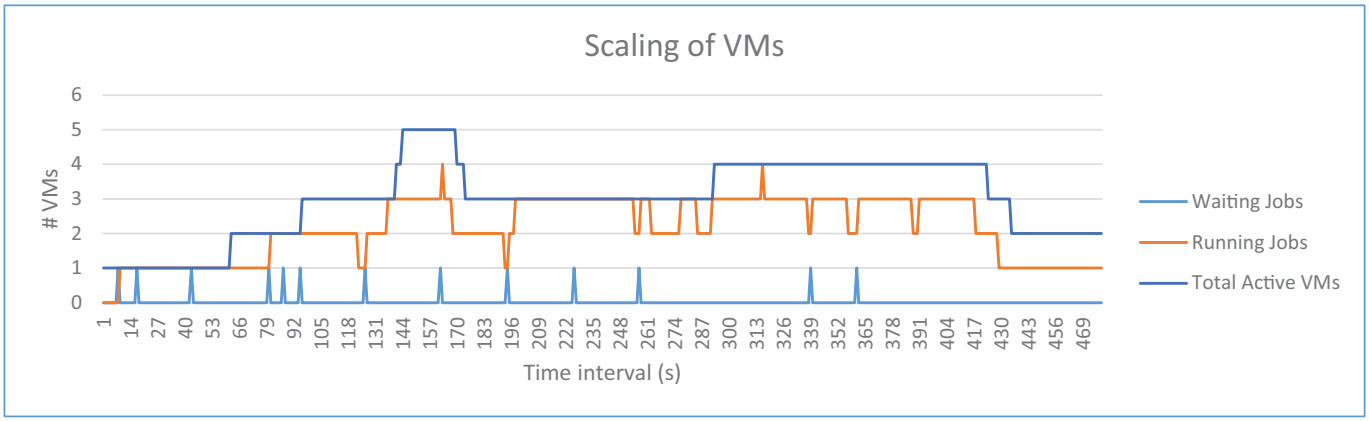

Fig. 4. The behaviour of CloudSicles resource scaling

TABLE I
SINGLE SLAVE PER ACTIVITY

Observed	Small	Medium	Large	Jumbo	Immense
files	2	20	100	1000	6739
total size (MB)	0.1	1.1	5.5	54.8	369.8
makespan (s)	13	33	148	405	1015

Inferred	Small	Medium	Large	Jumbo	Immense
time per file (s)	6.5	1.65	1.48	0.40	0.15
files / hour	554	2182	2432	9000	24000
EC2 cost/file (10^{-4} EUR)	1.80	0.45	0.41	0.11	0.04

TABLE II
THE CREATED VMS DURING EXPERIMENT 2. THE UTILIZATION IS CALCULATED BY MULTIPLYING THE AVERAGE RUNNING TIME OF A REQUEST (59 S) BY THE NUMBER OF REQUESTS A VM HAS PROCESSED. BECAUSE WE TAKE THE AVERAGE REQUEST TIME, THE UTILIZATION OF VM #39154 IS TOO HIGH.

VM ID	Running/ Charged Time	Requests Processed	Utilization	Costs
#39159	199	2	59%	€0,10
#39158	405	3	44%	€0,10
#39157	390	4	61%	€0,10
#39156	482	3	37%	€0,10
#39155	255	1	23%	€0,10
#39154	334	7	124%	€0,10

TABLE III
OVERVIEW OF VM USAGE METRICS FOR EXPERIMENT 2

Total running time:	2065 s
Average running Time	344 s
Total requests processed:	20
Average VM running time per job	17,21 s
Average utilization	58%
Total cost	€0,60
Cost per request	€0,03

capabilities of CloudSicle we needed to simulate a semi realistic workload over a certain timespan. We do this by generating 20 requests (of the medium workload) at random time intervals between 1 and 30 seconds.

Figure 4 shows the scaling of the resources over time. We see the total amount of VMs in the systems scales well with the number of jobs running. There is a nice buffer of available VMs, especially during periods where there are multiple VMs running at the same time. This would suggest that during consistent high request levels, CloudSicle will scale up accordingly. Scaling down is also shown in intervals 404 to 443. Another observation is the fact that jobs rarerely stay in the job queue of the Scheduler for more than a second.

Tables IV-C and IV-C show more data on the usage of the VMs. We see in Table IV-C that during this experiment the system created six VMs, that have an average lifespan of 344 seconds. The average utilization, ie how much time the VM is actually spending on jobs, of the VMs is around 58%. Because Amazon uses one hour increments, the costs for all the VMs are the same.

V. EVALUATION

In this section the evaluation of the experimental results as given in Section IV will be discussed. First the observed and inferred results of the two experiments we performed will be discussed. The second half of this section will deal with the extrapolation of these results.

A. Results

Following the results of our conducted experiments, it is interesting to discuss the implications of these results. The results of Experiment 2 show us one important trade-off in CloudSicle. While the scaling of resources works very good, it comes at the price of a high number of VMs (6 for 20 requests in total), for which we have to pay 10 cents for each new VM. This is primarily because of the timeout we chose to kill VMs after they are done. We chose that value to be low. If the charging-scheme of the cloud provider would be to charge for *actual* VM usage, this would be beneficial as we quickly remove redundant VMs. However, because Amazon charges by the hour, it would be much better to keep VMs

alive for the full hour. Fortunately, this can be implemented quite easily in CloudSicle.

Another observation is the fact that the actual combining of *.gif*s using Gifsicle is very fast, and does not use many CPU resources. It could be possible to easily run numerous Gifsicle processes parallel on one VM. Thus the question arises if WantCloud BV even needs a cloud infrastructure. A cloud infrastructure may simply be to much overhead for such a simple light weight application.

B. Extrapolation

As discussed in the introduction of this report, we assume the average CloudSicle job to contain 100 files on average. Given the calculated cost per file from Table I we can calculate the costs for 100 thousand, 1 million and 10 million users. For these calculations we assume have set the maximum amount of Slaves to 20. The result of these calculations is given in Table IV, we note that because we have not tested the splitting of jobs the increase with the amount of users is linear.

TABLE IV
EXTRAPOLATED CHARGE TIME & COST

Users (Millions)	Time (Hours)	Cost(EUR)
0.1	206	420
1	2056	4120
10	20556	41120

For all of the possible combinations of days, weeks, months and years in combination with millions of users we can also calculate the cost. If we filter out the impossible amounts of hours per timeframe we get the results of Table V. These calculation assume the worst case scenario where Virtual Machines get all of their work at once and the rest of the time the buffer Virtual Machine remains powered on. This causes maximum inefficiency in CloudSicle's buffering scheme.

TABLE V
EXTRAPOLATED COST PER DAY

Users (Millions)	Per	Cost(EUR)
0.1	Month	471
0.1	Year	1254
1	Year	4790

We note that the implementation of the SplitJob would drastically change the outcome of these calculations. If Virtual Machines are able to balance loads and continuously accept activities, the throughput of the system would be much higher (and therefore the cost would be much lower).

VI. CONCLUSION

CloudSicle has been our attempt at creating a well scaling Cloud service. This report has shown the fairly robust framework of CloudSicle with support for advanced allocation strategies, which are currently not available for the project. We have shown how our simple scheme performs and how it is well suited for processing large datasets per client. We have also shown how responsive CloudSicle is to new requests and the drawback of resource waste that comes with it. We note how CloudSicle is a robust framework and that it would benefit greatly from more advanced allocation strategies being used for it, as the current results leave room for improvement.

If WantCloud BV wants to continue with this proof of concept, we would like to improve CloudSicle in number of ways:

- Implement the SplitJob allocation policy, and evaluate its benefits compared to the default policy.
- Improve the performance of the downloading and uploading of files. One could for example compress the set of files the Client sends to the Slave.
- Creating a full API shell around gifsicle so the user can make use of all the functionality and options gifsicle offers.

ACKNOWLEDGMENTS

The authors would like to acknowledge the guidance of A. Iosup and D. Epema in exploring the world of Cloud Computing and offering the authors the chance to interact with the DAS-4 system. The authors would also like to acknowledge B. Ghit for his interactions when the authors were unable to log in to DAS-4 and also when OpenNebula accounts were needed.

APPENDIX A
TIME SHEET

This section contains the times spent on various things while developing and testing CloudSicle in Table VI. It also contains times for both experiments in Table VII.

TABLE VI
TOTALTIME

Activity	Hours
total-time	141
think-time	8
dev-time	108
xp-time	7
analysis-time	4
write-time	14
wasted-time	0

TABLE VII
EXPERIMENT TIME

Experiment	1	2
total-time	4.5	2.5
dev-time	0.5	0.5
setup-time	4	2

REFERENCES

[1] Kohler, E. *Gifsicle: Command-Line Animated GIFs*, http://www.lcdf.org/gifsicle/, Last retrieved October 30 2013
[2] Dean, J., & Ghemawat, S. 2008. *MapReduce: simplified data processing on large clusters*. Communications of the ACM, 51(1), 107-113.
[3] University of Amsterdam *DAS-4 Overview*, http://www.cs.vu.nl/das4/, Last retrieved October 30 2013
[4] Open Nebula *Java OCA API 2.0*, http://opennebula.org/documentation:archives:rel2.0:java, Last retrieved October 30 2013