WASP SECC Course: Cloud Module Assignment Report

Kristoffer Bergman, Per Boström, Shervin Parvini Ahmadi

June, 2017

1 Overview

This report briefly describes the service created in the Cloud module of the WASP course in Software engineering and cloud computing. The task was very time consuming since non of the group members had previous experience from similar work. Nevertheless, a scalable video conversion has been developed and is described in the following.

Git repository for the code is found at: `https://github.com/perbostrm/cloudassignment`.

An overview of the system architecture can be found in Figure 1. The different components involved are:

- **Workload generator**: Is used as a test engine to simulate external users. See Section 4 for more details.
- **Load balancer**: Was not implemented due to lack of time.
- **Frontend Web API**: Is used to receive requests from users and deliver a response. See Section 5 for more details.
- **Message queue**: Is used to route the requests to the available Video conversion nodes and maintain the request queue.
- **Backend Video conversion**: Is used to do the actual conversion of videos by reading the video from the storage. See Section 6 for more details.
- **Video Storage**: Is used to store available and converted videos.
- **Monitor**: Is be used to monitor entities relevant for the controller, by periodically gather information. The information is saved in .mat files, which can be used to plot the data in Matlab.
- **Controller**: Is used to make the service scalable. See Section 7 for more details.
- **Health-Checking**: Is used to perform health checking for the backend and frontend nodes.

2 Software and tech

The following is a list of the software and tech that is used in the system components:

- **Workload generator**: Python
- **Frontend Web API**: Python
- **Message queue**: RabbitMQ
- **Backend Video conversion**: FFmpeg with ffmpy
- **Video Storage**: Object storage in OpenStack
- **Monitor**: Python
- **Controller**: Python with python-novaclient (To interact with OpenStack).
- **Health-Checking**: Python with python-novaclient (To interact with OpenStack)

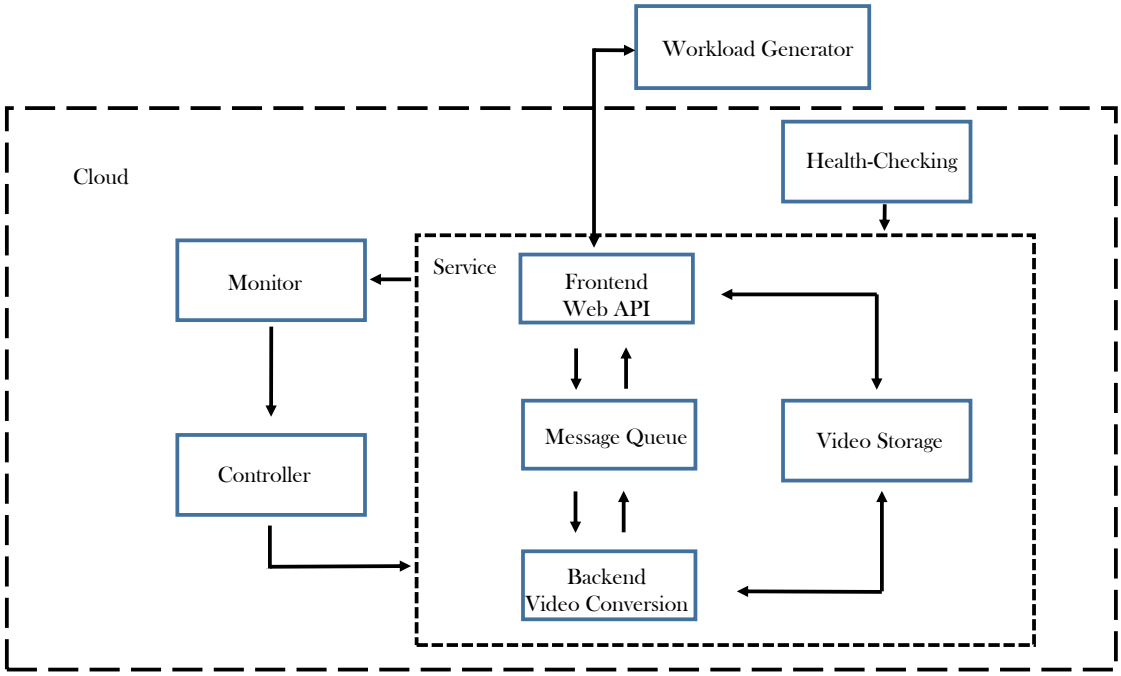

Figure 1: Overview of the system architecture.

3 Redundancy

Due to lack of time, we were unable to implement redundancy in the components with states (such as the Message Queue and the Monitor).

4 Workload generator

The workload generator is the test engine that simulates external users. It takes two parameters: the number of client converters and the average time between conversion requests. It starts the specified number of client threads, where each one represents a user. Each user randomly selects a video, sends a conversion request, waits for the completion, and then sleeps for a while (on average the specified time) before repeating the process.

The workload generator also measures the response time and stores it in a .mat-file.

5 Frontend Web API

The Frontend Web API is used to receive requests from users and deliver a response. The conversion request is handled according to the following steps

- The Frontend Web API receives a video conversion request, and sends this request to the Message Queue.
- Then, it waits for a response from the backend, which contains information of how to find the converted video in the storage.
- Finally, an "Accepted" response is sent to the user, where the information of how to find the converted video in the storage is included. If the backend does not accept the conversion request, an Error message is returned.

When a user is requesting the actual converted video, the Frontend Web API checks if the converted video (which has a unique name) is available in the Video Storage using python-swiftclient. If it can be found, it responds with a OK message, otherwise an Error messasge is returned.

6 Backend Video Conversion

The Backend Video Conversion node is used to perform the actual conversion of videos. When a conversion request is received from the Message Queue, the following steps are executed

- First, the node checks if the requested video exists in the folder of available videos. If it does, the node sends an Accepted response back to the queue together with an unique file name, which represent where the converted file will be stored. Otherwise, an Error response is returned and the conversion is aborted.
- The second step is to download the requested video from Video Storage and perform the actual conversion. For this, we used FFmpeg together with a Python wrapper called ffmpy.
- Finally, the converted file is uploaded to the Video Storage. Once this is done, the converted and downloaded videos are deleted locally.

7 Controller

The controller is used to scale up and down the number of VMs used for backend video conversion, depending on the workload. The controller is running at a frequency of 0.2 Hz

We used the simplifying assumption that all videos are of approximately the same size. This makes it possible to use the time from that a video conversion request is submitted until the video is converted as the control objective. Assume the average time it takes for one core to convert one video is given (including potential delays in the system), and denote it T_{conv}. With n_{queue} video conversion requests in queue and n_{vm} active VMs, the time T needed to convert all videos in the queue is given by

$$T = \frac{T_{\text{conv}} n_{\text{queue}}}{n_{\text{vm}}} \tag{1}$$

Thus, if the control objective is to keep the maximum time from a request is submitted until the video is converted below $T_{\max}$, the number of VMs needed is given by

$$n_{\text{vm}} > \frac{T_{\text{conv}} n_{\text{queue}}}{T_{\max}} \tag{2}$$

This is of course a simple approach which, if needed, could be extended to also consider videos of varying sizes and possibly reordering the queue such that for example small sized videos are prioritized over larger ones. By some simple tests, we got that $T_{\text{conv}} \approx 4$ seconds when we only had one request running at the time. We used $T_{\max} = 12$ as the maximum , which gave us

$$r_{\text{vm}} = \max\{n_{\text{lowest}}, \min\{n_{\text{highest}}, \frac{T_{\text{conv}} n_{\text{queue}}}{T_{\max}}\}\}$$

where $n_{\text{lowest}} = 2$ is the lowest amount of active video conversion nodes, and $n_{\text{highest}} = 10$ is the highest amount of active video conversion nodes.

To reduce the effect of quick, temporary changes in the queue, we have used a low-passed filtered reference signal. We have also implemented a minimum time to live for the video conversion nodes, since the time from creation to actual work is long (currently couple of minutes, see Figure 3).

8 Health-Checking

The Health-Checking node is used to make sure that the available Backend and Frontend nodes works. This is done by checking the Status flag in OpenStack. If a node has status Error, this node is deleted and then re-created again. This is done by using python Novaclient.

9 Experimental Results

Since we did not have time to implement the robustness for the system, the experimental evaluations were focused on scaling the service up and down depending on the workload. During the experiments, the workload generator was for convenience executed within the cloud.

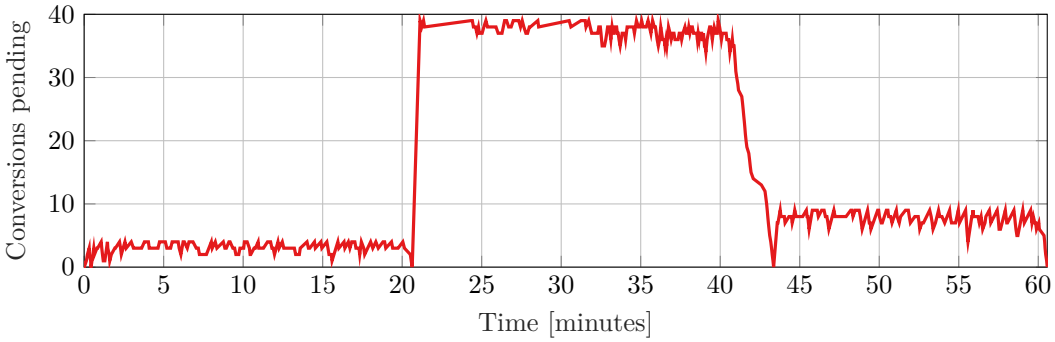

Figure 2: Number of pending conversions as a function of time. At $t = 20$ minutes, a sharp increase in the workload occurs and at $t = 40$ minutes the workload drops to a lower level.

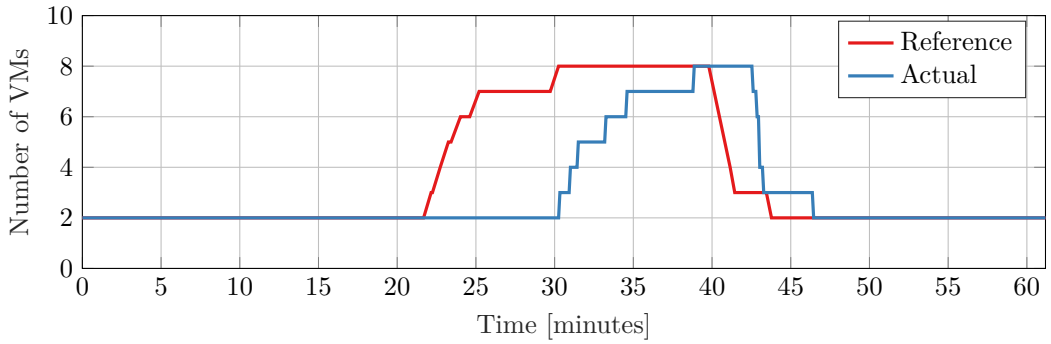

Figure 3: Number of virtual machines in the backend as a function of time. The red line is the reference signal from the controller, while the blue one represents the actual number of VMs connected to the message queue.

Figures 2–4 show data from a scale up and scale down scenario. The sharp increase in workload at $t = 20$ minutes leads to a substantial increase in the time it takes to handle conversion requests. To handle this, the controller increases the reference and more VMs are created. However, due to long boot times, it takes some time before the new VMs can start working. Also, we noticed that during the startup of new VMs, the ones that were already up and running seemed to slow down. The scaling down is handled in a similar manner; as the workload decreases, fewer VMs are needed and hence some of them are shut down.

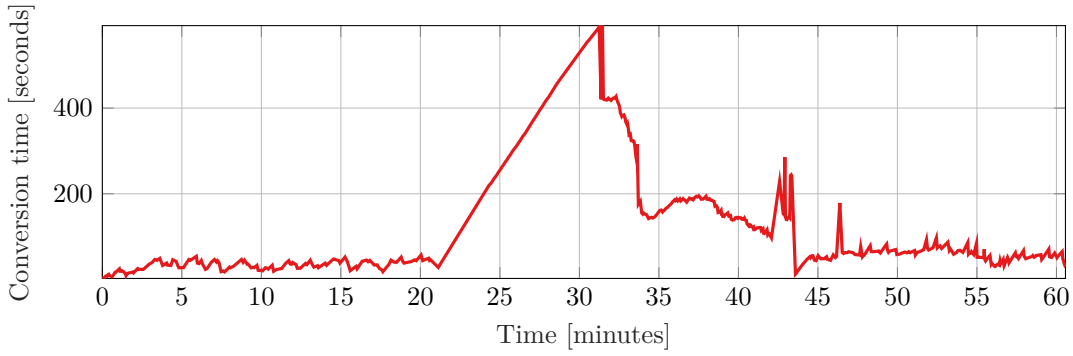

Figure 4: The time it takes to process a conversion request as a function of time.