Towards microservices architecture to transcode videos in the large at low costs

Olivier Barais, Johann Bourcier, Yérom-David Bromberg, Christophe Dion

Towards microservices architecture to transcode videos in the large at low costs

Olivier Barais* -†, Johann Bourcier*, Yérom-David Bromberg*, Christophe Dion†
* University of Rennes 1, IRISA, INRIA Bretagne Atlantique
† B-COM
Rennes, France
Email: barais@irisa.fr, johann.bourcier@irisa.fr, david.bromberg@irisa.fr, Christophe.DION@b-com.com

***Abstract*—The increasing popularity of videos over Internet, combined with the wide heterogeneity of various kinds of end users' devices, imposes strong requirements on the underlying infrastructure and computing resources to meet the users expectations. In particular, designing an adequate transcoding workflow in the cloud to stream videos at large scale is: (i) costly, and (ii) complex. By inheriting key concepts from the software engineering domain, such as separation of concerns and microservice architecture style, we are giving our experience feedbacks of building both a low cost and efficient transcoding platform over an *ad hoc* computing cloud built around a rack of Raspberry Pis.**

I. Introduction

In the last decades, we have witnessed a spectacular rise in popularity of video streaming over Internet. Industrials, such as `Cisco` or `Ericsson`, predict that video traffic will dominate other types of traffic by 2019, and will reach an astonishing 80% of the world's total internet traffic. The concomitant raising popularity of `Twich`, `YouTube` and `Netflix`, correlated with the emergence of various kinds of mobile devices, drives the massive increase in video consumption. As a result, we live in a video-oriented world. Such a phenomenon changes both our video consumption experience and habits. Video streaming is becoming a key benefit for any digital business. However, delivering video content to users imposes heavy constraints on the underlying infrastructure and computing resources to meet the needs and the end users' expectations. In fact, streaming videos to consumers' devices, requires to encode each video in various formats to meet devices' specificities. In practice, it is not realistic, nor possible to store videos in all possible formats at the server side. To alleviate this burden, `HLS` and `MPEG-DASH`, gain momentum to become the de facto standards to deliver a dynamic adaptive video streaming [1], [2]. Videos are splited into a sequence of segments, made available at a number of different bitrates (i.e. at different quality levels), so that clients can automatically download the adequate next segment to be played, based on both their characteristics and their network conditions. However, still, video transcoding is cumbersome: it requires a massive amount of server side resources, and thus an infrastructure that scales adequately.

The emergence of cloud computing, combined with adaptive bitrate delivery, paves the way for new solutions that scale better. The fact that segments from the splited video are independent from each other enables to setup various patterns to spread over a set of servers the segments to be transcoded. Hence, many research works have been done to provide, as much as possible, the most adequate algorithms, from different optimization strategies, to schedule tasks, to transcode a sequence of segments, on a group of interconnected servers [3], [4], [5]. As a result, according to the transcoding workflow, algorithms optimize either CPUs consumption, network bandwidth consumption, transcoding time, customers' quality of experience or a mix of these aforementioned properties though the use of heuristics or empirical studies. Most of the existing solutions (from either academia or industry such as `Netflix`), leverage on Infrastructures as a Service (IaaS) clouds, such as Amazon Elastic Compute Cloud (EC2), to provision and instantiate on the fly the adequate numbers of Virtual Machines (VMs), dedicated to the transcoding of video segments depending on the incoming load. From this context, whatever the chosen scheduling algorithms, it appears clearly that streaming video in the large comes with an inherent significant cost to build a video transcoding cloud computing platform whatever the considered scaling methodology: vertical scaling to increase/decrease instance capacity, and/or horizontal scaling to add/remove instances [6], [7].

In this paper, we provide feedback experiences about the use of an *ad hoc* cloud computing platform built from a farm of Raspberry Pis to promote low cost but efficient video transcoding. The inherent resources constraints of Raspberry Pis give us the opportunity to revisit how to design and implement the commonly used transcoding workflows. First, using VMs to scale video transcoding is not the most adequate approach as it requires the use of hypervisors, or virtual machine monitors (VMMs), to virtualize hardware resources. VMMs are well known to be heavyweight with both boot and run time overheads [8], [9] that may have a stronger impact on small bare metal servers such as Raspberry Pis. Consequently, we promote the use of containers, such as Docker, as a lightweight alternative, that startup as fast as normal processes. Further, the transcoding workflow involves complex interleaved operations such as *splitting*, *merging*, *scheduling*, *storing*, *transcoding*, and *streaming*. To cope with this inherent complexity while fostering and increasing evolutivity, maintainability, scalability, we are designing our approach arounds the best practices in software engineering, learnt, in particular, from the experience

of microservices architectural style[1]. As a result, we apply a separation of concerns to isolate each operations into a graph of microservices isolated into lightweight containers. Since, microservices have the inherent ability to be isolated from each other, we are able to spawn as much as expected containers to provide an agile development and continuous delivery. Consequently, our approach enables us to experiment in an easy manner how different transcoding workflow may scale depending on the deployed microservice architecture, and the underlying strategies to split, merge, schedule, store, and transcode video streams.

Our contributions are as following:

- We have designed an *ad hoc* cloud computing platform built around a rack of 16 interconnected Raspberry Pis.
- We have revisited the traditional transcoding workflow usually encountered in cloud computing platforms from a software engineering perspective. Specifically, we have applied key software engineering concepts such as separation of concerns and microservices architectural style.
- We have demonstrated that this architecture style eases the design of a scheduler that can decide dynamically if it uses GPU or CPU to transcode a video chunk.
- We are providing a suitable testbed to elaborate a low cost platform that is able to transcode efficiently videos at large scale.
- We perform a thorough performance evaluation of our platforms

The remainder of this paper is structured as follows. Section II provides a general overview of the proposed approach. Section III details the technologies that are integrated to implement this approach. Section IV evaluates the proposed approach regarding the performances and the complexity of the solution. Section V discusses a research roadmap and concludes this experiment.

II. Approach

Each operation involved in our transcoding workflow is instantiated as a microservice that is deployed on a lightweight container (See Figure 1). Further, a microservice can be spawned several times across the cloud, i.e on different machine for either reliability and/or scalability. For instance, a *splitter* microservice splits a video files into a set of chunks; the latter are then forwarded to a *scheduler* microservice to distribute the chunks among *transcoder* microservices spawned multiple times across the cloud. The aim of the *scheduler* microservice is to perform an accurate load balancing of transcoding jobs among the multiple instances of *transcoder* microservices. There exists a huge set of different strategies to schedule jobs dedicated or not to multimedia streaming [10]. Providing a new kind of scheduler is not currently, in this paper, our key concern. However, in our approach, as each operation is isolated from each other, we can change in a smooth manner the scheduling strategy.

[1]http://martinfowler.com/articles/microservices.html

Currently, our default *scheduler* microservice implements an enhanced round robin strategy, also named *first fit* [10]. According to the incoming chunks, the scheduler selects the first and/or the most available *transcoder* microservices to dispatch the chunk. In fact, each *transcoder* microservice has a FIFO buffer to store incoming chunks. Chunks are poped one by one by the transcoder to be transcoded. If no transcoders are available, the cloud is then in a saturated state, and the *scheduler* microservice will not dispatch any new chunks until one of the transcoder's buffer is drained. The *scheduler* microservice is aware of the availability state of the different transcoders through the *storage* microservice that acts as a distributed key/value storage, which uses a gossip protocol to propagate, for instance, updates of availability states among microservices of the cloud. Further, the *scheduler* microservice may interact with a *monitor* microservice to get resources usage such as CPU, RAM and network statistics, related to each transcoder, to refine its load balancing policy. Once a transcoding job is finished, the *merger* microservice is notified, and then reassembles incrementally the encoded chunks that are finally streamed from the *streamer* microservice.

We have setup an automated deployment process. From a bench of tools and scripts, all the aforementioned microservices are continuously and smoothly deployed onto our *ad hoc* cloud computing platform (See Figure 1 ❷,❸). Microservices are thus automatically instantiated in lightweight containers on the underlaying 16 bare metal servers (Figure 1 ❹) without requiring any specific configuration efforts. Consequently, as soon as a user sends a request to get a specific movie from the cloud (Figure 1 ❺), the transcoding workflow is started.

Currently, we are not doing any caching as our main objective is to evaluate the on the fly transcoding feature of our approach. However, adding a caching mechanism is as easy as adding a new caching microservice in our architecture.

Finally, our approach may be seen as a particular implementation of a *map/reduce* model [11]. In fact, the *splitter* microservice acts as a preliminary phase to feed the available mappers. The *map* operation consists then in transcoding chunks, just as the *transcoder* microservices do. Finally, the reduce operation is similar to the *merge* microservices jobs. However, our approach enables a greater flexibility in the workflow complexity as promoted by the microservice architectural style.

III. Implementation

Our implementation relies on lightweight container technologies and low cost hardware, namely `Docker` and `Raspberry Pi 2` (RPi). We have setup an *ad doc* cloud computing platform to take benefits from a farm of RPi. One key characteristics of our specific cloud is to enable us to both deploy and run containers in a smooth and easy manner whatever the number of RPi involved. Moreover, we have built a dedicated workflow that is able to take benefits from the underlying cloud to transcode videos in a distributed manner. To reach this aim, we have containerized various video transcoding software commonly used. Finally, we have

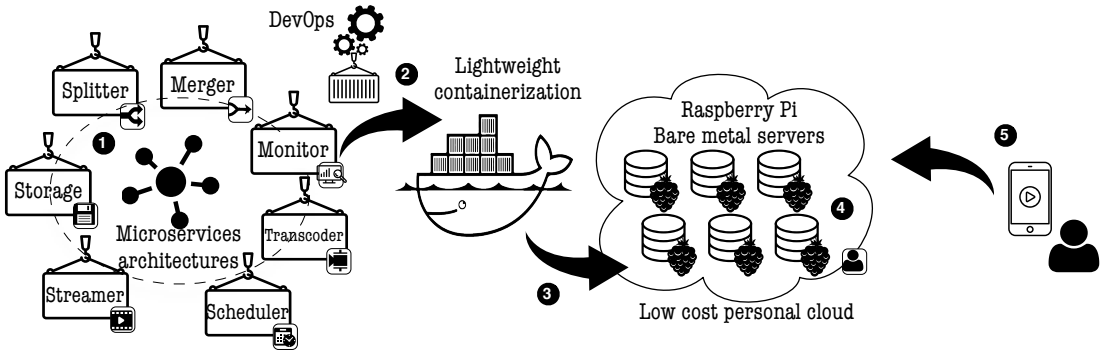

Fig. 1: Approach

fine-tuned scripts to use a set of tools to both understand how to optimize our video transcoding workflow, and get good performances that scale.

A. Setting up the ad hoc cloud computing platform

Our cloud is in charge of scheduling, deploying and executing all tasks relative to video transcoding in a distributed way. The key challenge is to optimize the cloud in such a way that its global resource usage is uniformly saturated. The following tools are used to setup our clouds.

Docker: In this work, `Docker`[2] is used as our underlying lightweight container system. It provides us a common way to package and deploy microservices. Particularly, it provides a high level API to manage hardware resource allocated to each microservice. It enables us to place and isolate each process on a specific core. We leverage this particular feature to indicate placement constraint for each task. For instance, the hardware encoding task must run on a GPU, and not on one core of a classical CPU.

Docker Swarm: Each microservice is containerized; it is encapsulated inside a docker image ready to be deployed and run on top of our `Raspberry Pi` farm. To deploy microservices, `Docker Swarm`[3] is used as it provides native clustering facilities. Its related API, for managing deployment of containers, enables us to control how to schedule the deployment of microservices over the pool of `Raspberry Pis`. In particular, `Docker Swarm` may follow different strategies to schedule the deployment of docker images on docker hosts: (i) *random*, which randomly allocates a host, (ii) *spread*, which chooses the host containing the minimum number of containers, and (iii) *binpack* which places as much container as possible on the minimum number of host. In addition, `Docker Swarm` uses filter to select eligible hosts for running a specific docker image. In our context, we use the *spread* strategy to choose a specific host, and we use filters to put constraint on the devices capable of hosting specific transcoding task.

Go-docker: One key shortcoming of `Docker Swarm` is that it does not enable to express precedence among tasks. To overcome this issue, `Go-Docker`[4] is used to schedule all tasks over the `Docker Swarm`, while respecting precedence constraints and optimizing resource usage. `Go-Docker` can be seen as a batch cluster management tool. It uses inherently `Docker Swarm` to actually dispatch docker images on remote nodes. Hence, all the tasks (such as chunk copy, chunk software encoding, chunck hardware encoding, merge, ...) that have to be executed with precedence and location constraints are managed by `Go-Docker`.

Consul: `Consul` serves a triple purpose. First, it acts as a service discovery for the `Docker Swarm` cluster itself. Second, it provides failure detection, service registration and service discovery functionality for all containers launched on the cluster. Thirdly, it provides a flexible distributed key/value storage service for dynamic configuration, feature flagging, coordination, leader election. For instance, in our scenario, `Consul` allows us to know when a task is over.

B. Video transcoding workflow

Our video transcoding workflow (depicted in Figure 2) is composed of several tasks:

1) a *splitter* to split the video in several small video chunk.
2) a *chunk transfer* task to transfer each chunk to a set of targeted host
3) a *scheduler* takes the decision to start the transcoding process on a specific node based on runtime information.
4) a *video encoding* task with two different implementations: one for software encoding and another one for hardware encoding.
5) an *encoded chunk transfer* to transfer the encoded chunk back.
6) a *merger* task which gather all encoded video chunk and assemble them incrementally.
7) a *streamer* task which takes the output of the merger task to stream the newly encoded video.

The RPi has an under-powered CPU but a powerful GPU. Consequently for encoding the videos chunk, we uses both in parallel: the software encoding on top of the CPU and the hardware encoding on top of the GPU. Consequently, we rely on the following software libraries that are packages as docker containers to implement all the above mentioned tasks:

a) FFmpeg: FFmpeg is a project that groups libraries and programs for handling multimedia data. FFmpeg includes h264 software encoder. We use it to enable chunk software

[2]https://www.docker.com/
[3]https://docs.docker.com/swarm/
[4]http://www.genouest.org/godocker/

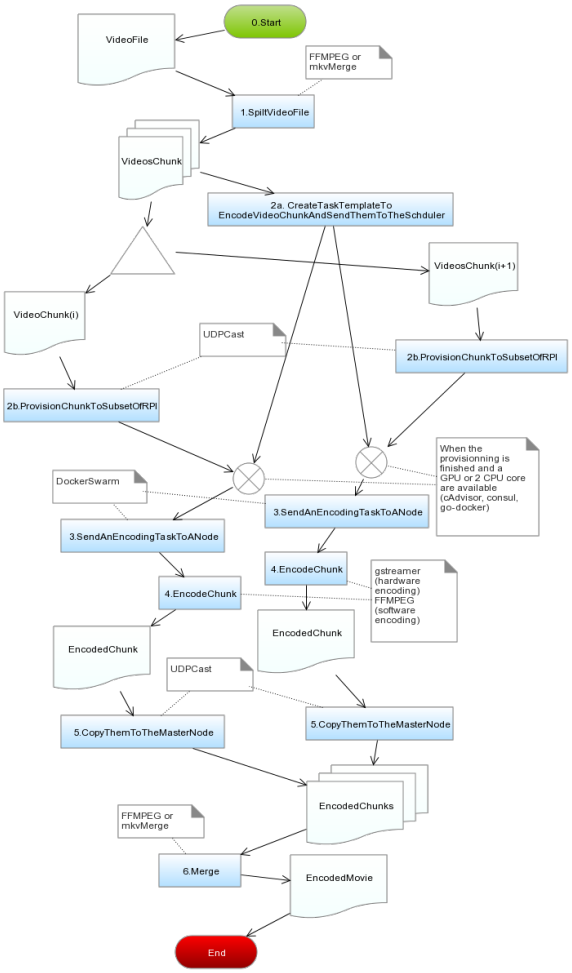

Fig. 2: Transcoding flow chart

encoding on available core of the RPi cluster. We also use it for splitting and merging videos chunks.

b) MKVToolNix: MKVToolNix is a set of tools to create, alter and inspect Matroska files. We use it to split (create chunks) and merge Matroska videos files (From our experiment, FFmpeg does not correctly split and merge Matroska videos files).

c) OpenMAX: The RPi has a very under-powered CPU but a powerful GPU. Consequently, it is interesting to use also the GPU for encoding some videos chunks. To access to the GPU, it exists an open standard (OpenMAX) that are designed for low capability devices. Broadcom have implemented one of the specifications (OpenMAX IL), and their chip is used in the RPi. Even if OpenMAX is an extremely difficult API to work with, it exists a GStreamer module that provides an implementation of an h264 encoder.

d) GStreamer: GStreamer is a multimedia library for constructing graphs of media-handling components. It supports range from simple Ogg/Vorbis playback, audio/video streaming to complex audio (mixing) and video (non-linear editing) processing. In our experiment, Gstreamer is used to manage the h264 hardware encoding based on OpenMax librairies.

e) UDPCast: . For spreading the videos chunks on all the RPi, we use a multicast file-transfer tools nammed udpcast. UDPcast sends data simultaneously to many destinations on a LAN. In out case, we send each chunk to p nodes where p is upper than 1 and lower than n (the number of RPi) in order to provide fault tolorance and improve the freedom where we could place the encoding task.

Based on this technical stack, our workflow is as follow. We use $MKVToolNix$ or $FFmpeg$ to create chunks. Then, we use $udpcast$ to spread a first set of chunks to the nodes and we send all tasks to execute to go-docker that will decide when the task can be started. Next, we send a first set of task to encode chunk. We use in parallel 3 core of each RPi with $FFmpeg$ to use a software encoder, and $gstreamer$ to use in parallel an hardware encoder for encoding another chunk. During this first, set of chunk encoding, we send the next set of chunk to the nodes. When a chunk is encoded, we deploy a task to copy back the encoded chunk to the master nodes. When all the chunks are encoded and copied back to the master nodes, a last task performs the merging using $MKVToolNix$. During all the process, go-docker is used to batch the tasks.

C. Analysing performance

In the context of this work, we also use a couple of tools to understand and optimize the performance of our distributed video transcoding infrastructure. We therefore used the 3 following tools.

cAdvisor: cAdvisor (Container Advisor) provides container users an understanding of the resource usage and performance characteristics of their running containers. It is a running daemon that collects, aggregates, processes, and exports information about running containers. Specifically, for each container it keeps resource isolation parameters, historical resource usage, histograms of complete historical resource usage and network statistics. CAdvisor allows us to precisely monitor the current execution of our workflow. In this work, we used cAdvisor to improve and optimize the performance of our scheduler.

InfluxDB: InfluxDB is a distributed time series database. cAdvisor only displays realtime information and doesn't store the metrics. We need to store the monitoring information which cAdvisor provides in order to display a time range other than realtime.

Grafana Metrics Dashboard: The `Grafana Dashboard` enables to pull all the pieces together visually. `InfluxDB` and `Grafana` are used for post mortem analysis of our workflow execution.

IV. Evaluation

The goal of this evaluation section is to provide feedback experiences about the use of an *ad hoc* cloud computing platform built from a farm of Raspberry Pis to promote low cost but efficient video transcoding. To this end, we have evaluated three main axis regarding our implementation and deployment:

Device	*Number*	*Unit Price*	*Total*
Rapsberry Pi 2	16	35$	560$
Switch	1	80$	80$
Alimentation	4	10$	40$
8Gb SDCARD class 10	16	5$	80$
Ethernet cables	16$	1$	16$
Total for a small cluster			776$

TABLE I: Cost to build an *ad hoc* personal cloud

- the **cost** of our solution, both regarding **financial investment and energy consumption**.
- the intrinsic **encoding performance** of our specific deployment setup.
- the **development effort** required to setup such platform for video transcoding.

A. Cost of the solution

Our specific experimental setup is composed of a farm of 16 Raspberry Pis 2, a 24 ports Gigabit switch, a set of ethernet cables and the corresponding power cables to provide the required energy. This setup is shown on the figure 3. Our experimental setup offers a computing infrastructure of 16 GPU and 64 ARM cores. As shown in Table I, the hardware cost of our small cluster is 776$. This cost is comparable to the price of a working station configured with a state of the art CPU : an intel I7 CPU.

To evaluate the power consumption of our cluster, we have measured the consumption of each hardware device separately using a smart plug which monitor energy consumption. Each raspberry Pi in our setup consumes roughly 4W during the experiment. The switch which connects all raspberry pi together consumes itself 15 watts during the experiment. In total, our farm of 16 Raspberry Pi with the switch consumes about 80 Watts when used at full power. This power consumption is comparable to the power consumption of an i7 5775C which consumes on average 99 watts when performing a x264 encoding task [5].

Thus the price and the power consumption of our small scale cluster of raspberry pis is comparable to a typical workstation using a high performance processor with a comparable amount of memory.

[5]taken from the hardware test conducted here http://techreport.com/review/28751/intel-core-i7-6700k-skylake-processor-reviewed/5

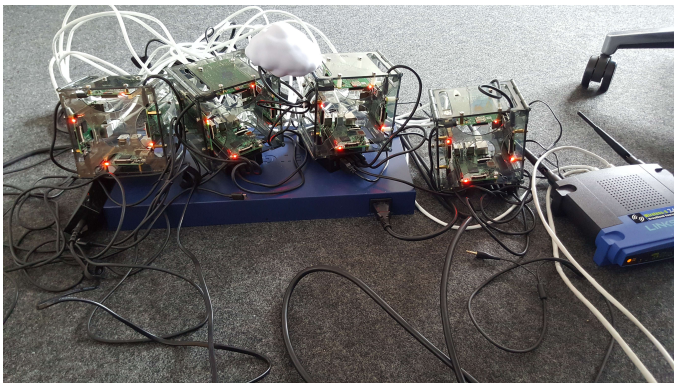

Fig. 3: Our experimental setup

B. Encoding Performance

In this section, we compare the performance of encoding a video both using a workstation (featuring an Intel(R) Core(TM) i7-5600U CPU @ 2.60GHz, 16 Gb of memory, running on Linux Ubuntu) and our cluster of 16 Raspberry Pi 2. Our experiment consists in encoding a H264 video file into another H264 video file with the ”High profile”. The input video and output video have a resolution of 1280*688 with 24 images/seconds each. In this section, each experiment was repeated 10 times, and we present average value of these 10 runs.

Our first experiment compares the time needed to encode a small video chunk of 2 min and 30 seconds both on the workstation and a single raspberry pi 2 to setup a base line comparison. On the raspberry pi 2, we performed two different experiments. The first one performs software encoding leveraging the 4 available cores on the ARM processor. This experiment is done using the ffmpeg software. The second experiment performs hardware encoding leveraging the GPU available on the raspberry Pi. This experiment is done using the Gstreamer omx software. Table II presents the result of this experiment. These results shows that using software encoding the Raspberry Pi2 takes 12 times longer than the workstation used in our experiment, while the hardware encoding takes 4.3 times longer.

Device	*Encoding*	*Time in second*
Rapsberry Pi 2	Software	1601.5 s
Rapsberry Pi 2	Hardware	554.6 s
Workstation (i7)	Software	126.9 s

TABLE II: Time required to encode small video chunks

Our second experiment evaluate the performance of the full farm of raspberry Pi2 against the workstation. In this experiment, we leverage the full power of the raspberry pi 2 cluster, thus using both hardware and software encoding in parallel. This experiment uses exactly the same setup as the first experiment, but the input video is longer : 25 minutes. Tables III shows the results of this experiment. These results show that our cluster of Raspberry pi 2 outperforms the workstation by achieving this task 2.4 times quicker.

Device	*Time in second*
Farm of Rapsberry Pi 2	530,2 s
Workstation (i7)	1281 s

TABLE III: Time required to encode long video chunks

The results shown in this section highlights the very good performance of our approach to provide a low cost, low power efficient video encoding infrastructure. Indeed, the farm of 16 raspberry Pi 2 is 2.4 times quicker for encoding a video than the same task on our comparison workstation.

C. Setup complexity

To complement our performance, cost and energy evaluation, we have evaluated the development effort which was required to setup such an infrastructure.

The first part was to setup the infrastructure on each raspberry Pi and the docker swarm. Therefore all the following commands have to be repeated on each raspberry (16 times for our setup): 1 command to flash the system, 1 command to copy ssh public key, 1 command to upgrade , 1 command to copy the configuration file for the cluster (cluster-lab configuration file), 1 command to change the docker configuration to provide privileged mode for docker (require to access to the GPU from the docker), 1 command to restart docker daemon and the cluster lab daemon.

After setting up the docker swarm, we had to prepare all docker images for each task of our video encoding infrastructure. Several tools used in our setup are mainstream and therefore docker images was available: consul, cAdvisor[6] and dockerui. Some images that were needed for our experiment were not available. We have thus created the docker image for go-docker and a template image for all our video encoding tasks. Finally, we completed our experimental with a shell script which discusses with Go-docker to run the batch of command. This shell script comprises 28 line of codes [7].

V. Conlusion and Research roadmap

We have designed our experiment to setup a video encoding infrastructure on a farm of low cost, low energy devices, through a bunch of open source, off the shelf software libraries. We show that the use of a micro-services approach ease the design of an ad-hoc map-reduce transcoding process that can leverage the computation power of low-cost board like RPi with good performance results. However, we have noticed several lacks, or limitations of current off the shelf tooling to simplify the deployment and optimize the operations of this video encoding infrastructure. This section therefore identifies opportunities for improving state of the art software components.

A. Tooling for non-expert

Our long term goal is to offer a framework which leverages existing tooling to easily deploy a specific video encoding process on a farm of low cost, low power computing devices. To reach this goal, we notice the lack of suitable tooling for non-expert to design the video encoding process and how this process can be automatically parallelized and deployed on a distributed system.

We also noticed a lack of specific tooling for non-expert to add specific treatment on demand during the video encoding process. For example, one may want to add a specific indexation mechanism during the video encoding process, or incrust a second video in the first one depending on user request. These specific examples show the need for an architecture which can automatically plug third parties software working on the video encoding.

B. ensure correctness of the resulting video

When automatically designing and deploying such a video encoding infrastructure, it is really important to perform test to ensure that the resulting video conforms to the expectations. We notice a lack of ready to use, off the shelf software libraries to automatically verify that the resulting video is conform to the user expectation.

C. Optimization of the distributed video encoding system

We noticed several opportunities for improving the performance of our distributed system to saturate the resource usage of each computing device involved in our cluster. On one hand, it would be beneficial to predict the duration for encoding each video chunk in order to optimize placement of each video task and chunk data on RPi. On the other hand, it would also be beneficial to be able to start video encoding before the first chunk is entirely delivered to the first computing device.

[6]https://github.com/RobinThrift/raspbian-cadvisor

[7]All the scripts are available online http://olivier.barais.fr/blog/

Acknowledgements

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreements No. 611337, the HEADS project, FP7-ICT-2013-10, from the CHIST-ERA under project DIONASYS, and from the French National Research Agency (ANR) under grant ANR-14-CHR2-0004.

References

[1] M. Grafl, C. Timmerer, H. Hellwagner, W. Chérif, D. Negru, and S. Battista, "Scalable video coding guidelines and performance evaluations for adaptive media delivery of high definition content," in *2013 IEEE Symposium on Computers and Communications, ISCC 2013, Split, Croatia, 7-10 July, 2013*, 2013, pp. 855–861.

[2] I. Sodagar, "The MPEG-DASH standard for multimedia streaming over the internet," *IEEE MultiMedia*, vol. 18, no. 4, pp. 62–67, 2011.

[3] J. Guo and L. N. Bhuyan, "Load balancing in a cluster-based web server for multimedia applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 11, pp. 1321–1334, 2006.

[4] A. Ashraf, "Cost-efficient virtual machine provisioning for multi-tier web applications and video transcoding," in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, May 2013, pp. 66–69.

[5] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou, "Apollo: Scalable and coordinated scheduling for cloud-scale computing," in *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, 2014, pp. 285–300.

[6] B. C. Tak, B. Urgaonkar, and A. Sivasubramaniam, "Cloudy with a chance of cost savings," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 6, pp. 1223–1233, Jun. 2013.

[7] E. Walker, "The real cost of a cpu hour," *Computer*, vol. 42, no. 4, pp. 35–41, 2009.

[8] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast distribution with lazy docker containers," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, Feb. 2016, pp. 181–195. [Online]. Available: https://www.usenix.org/conference/fast16/technical-sessions/presentation/harter

[9] J. Zhu, Z. Jiang, and Z. Xiao, "Twinkle: A fast resource provisioning mechanism for internet services," in *INFOCOM, 2011 Proceedings IEEE*, April 2011, pp. 802–810.

[10] J. Guo and L. N. Bhuyan, "Load balancing in a cluster-based web server for multimedia applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 11, pp. 1321–1334, 2006. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2006.159

[11] A. Garcia, H. Kalva, and B. Furht, "A study of transcoding on cloud environments for video content delivery," in *Proceedings of the 2010 ACM Multimedia Workshop on Mobile Cloud Media Computing*, ser. MCMC '10. New York, NY, USA: ACM, 2010, pp. 13–18.