Migrating live streaming applications onto clouds: challenges and a CloudStorm solution

Huan Zhou*, Spiros Koulouzis*, Yang Hu*, Junchao Wang*, Alexandre Ulisses‡, Cees de Laat* and Zhiming Zhao*
*Informatics Institute, University of Amsterdam, Amsterdam, Netherlands
‡MOG technique, Portugal
Email: *{h.zhou, s.koulouzis, y.hu, j.wang2, delaat, z.zhao}@uva.nl, ‡alexandre.ulisses@mog-technologies.com

***Abstract*—Live TV production, due to its distributed nature, requires broadcasters to deploy equipments and human resources to several different places. This increases production costs. The traditional method through outside broadcasting vans is expensive. Migrating this type of application onto clouds is a promising method to reduce the cost. However, the Quality of Experience (QoE) can hardly be assured because of the cloud performance uncertainty. Auto-scaling the infrastructure based on dynamic workloads at runtime is also difficult. The feasibility of CloudsStorm framework, which is the core component of SWITCH (Software Workbench for Interactive, Time Critical and Highly self-adaptive Cloud applications) workbench, handling the lifecycle of the time critical application development and performance monitoring, infrastructure planning and provisioning, etc., is demonstrated with the aspect of live events broadcasting. It makes a significant attempt to fill the DevOps gap when migrating applications from legacy systems onto clouds. A live streaming application is demonstrated as a use case example.**

***Index Terms*—Cloud, DevOps, broadcasting, live streaming, time-critical applications.**

I. Introduction

The production of live TV events by its very nature requires very strict requirements: delivering video and audio with as little delay as possible while maintaining the quality and security requirements that the television industry requires to ensure the maximum quality of experience (QoE) to viewers. The current method for outside live production is based on remote production based on mobile production trucks, or as it is called in the media sector, Outside Broadcasting (OB) vans [1] [2]. An OB van can cost several million dollars [1], mainly because of the large amount and diversity of equipment it contains. In many cases it may be necessary to use several units, mainly because of the distributed nature of live events, such as the coverage of an electoral event. This need, as well as the satellite connection that allows the connection between the OB van and the television studio, raises the costs and complexity of television production [3].

Although a modern production studio already extensively uses the IP protocol, its usage is mostly limited to archival of multimedia materials so that they are available throughout the production chain [4]. In the same way, this infrastructure allows to serve viewers with videos on demand (VOD) services such as Netflix and Hulu [4]. Despite the recent evolution, this is only a small part of the workflow in television production.

The evolution of virtualization's technologies on the cloud and the efforts developed in order to provide solutions of virtualized, elastic, controllable cloud environments leverages the adoption of this technologic stack for time-critical applications, such as a live TV production. However the usage of such type of technologies is still in its infancy [5], since there is still a DevOps (software Development and Operations) gap for the application to leverage the cloud more easily and efficiently, especially for handling the operations of auto-scaling, failure recovery, etc.

In this paper we focus on supporting live streaming applications using clouds, and highlight challenges during the DevOps lifecycle, including customizing, provisioning, and runtime managing virtual infrastructure based on the time critical constraints of live streaming. The solution through leveraging CloudsStorm[1] framework is carried out. The research is performed in the context of EU H2020 SWITCH project[2]. CloudsStorm is adopted as the core engine of the SWITCH workbench. In the rest of the paper, we firstly discuss challenges of the live streaming requirements, and then present the basic architecture of CloudsStorm framework with some key techniques implemented by the framework. After that, a use case is used to demonstrate how to leverage it to develop, deploy and run the application based on the current implementation of SWITCH [6] workbench.

II. Live Streaming Applications and Challenges

A. *Overview of live streaming applications*

For the production of live TV events in a distributed way, a live streaming application in the cloud is presented, supported by the transmission of video over IP and that allows the director, through a Web App, to perform actions such as changing the camera, selecting the number of input streams and choosing which output feed to obtain.

By using the cloud scalability capabilities, for example, it is possible to handle the video transcoding in the cloud or to adapt the number of streams that connect simultaneously to the application. Performing software-based operations, such as transcoding or cloud switching, which were once performed by hardware, enables virtualization to be produced not only for

[1]https://github.com/zh9314/CloudsStorm
[2]http://www.switchproject.eu/

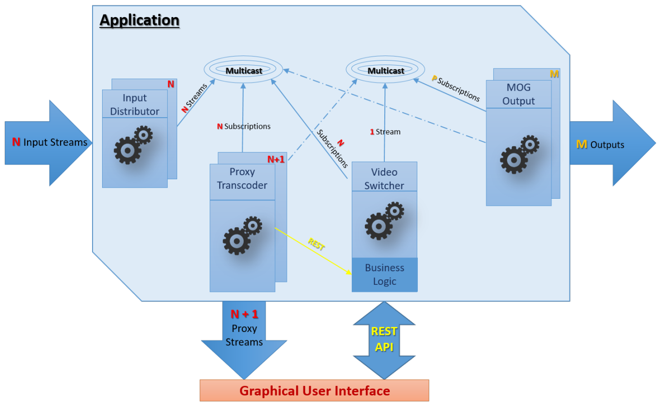


Fig. 1: Architecture for live streaming applications

a distributed environment but also for a flexible and adaptive environment.

The distributed application must be able to receive video streams and serve (low-resolution) proxy versions for the Web App, while dealing with the synchronization of streams and the coherence that is inherent in the output. When served to the end user (broadcasters) as SaaS, it allows constant updates transparently to the user, while taking advantage of the flexibility of the cloud.

The present application is composed of four blocks shown as Figure 1, each with a distinct and very specific function. Each block corresponds to a node in the network and, although they have different functions, the level of abstraction that they offer to the network and the communication between nodes are identical among all of them.

B. *Components and requirements*

According to Figure 1, functions of different components are as follows.

Each Input Distributor node is responsible for receiving an input stream, decompressing it, and delivering it, by multicast, generating the resulting media flows. In this case, the main relevant nodes are the Video Switcher and the Proxy Transcoder. Each Proxy Transcoder is responsible for transcoding the pair of media flows it has subscribed to, generating a proxy version and making it available externally, for example for a Web App.

The Video Switcher must subscribe to the multicast addresses that the Input Distributors are providing, store in memory the data it receives, and is served by multi-casting the Flow that Business Logic determines. It is necessary to save data in memory due to the delay introduced by the various network transmissions and the transcoding process performed by the Proxy Transcoder. Despite this, [7] suggests another IGMP-based switching technique, which, by itself, does not guarantee frame accurate switching.

Each Output node receives, by multicast, video flow from the Video Switcher and delivers them abroad, in a single stream, with a specific encoding and for a specific platform. This means that there may be multiple Outputs, including, for example, an Output that delivers a stream with the same Input characteristics as to provide cascading scenarios.

For each input stream an Input Distributor is provisioned; therefore, there is a 1: 1 ratio. This means that since N is the number of input streams entering the cloud, N input streams imply N Input Distributors. Due to the demultiplexing that the Input Distributor performs, N Flows are obtained, since each input stream gives one flow of video.

The relationship between an Input Distributor and a Proxy Transcoder is also 1: 1. That is, for N Flows from the Input Distributors set, there are N subscriptions made by N Proxy Transcoders, each subscribing to the corresponding Video. In addition, there is one additional Proxy Transcoder, responsible for serving the Video Switcher result, so the final Proxy Transcoders ratio is $N + 1$.

Let M be the number of Output nodes, which is independent of N, considering that there are P subscriptions to the switching result achieved by the Video Switcher. Since an Output can, for example, be a SD version of a program, the number of subscriptions P is not necessarily M. Likewise, the Output distributor can subscribe to Q flows directly from the Input Distributor, for the purpose of completing a transcoding string, for example.

Therefore, it should be noted in Table 1 that, for each set of nodes, the quantity that is supplied and the relationship between the incoming Flows and the Flows that are originated are gathered.

C. *Time critical challenges*

The implementation of this kind of system faces several challenges, as the system must:

- Collect and process the camera video data in nearly real time;
- Predict the potential increase of load on the warning system when public users (customers) increase;
- Operate reliably and robustly throughout its life time;
- Scalable when the deployment of cameras increases.

The development of such applications is usually difficult and costly, because of the high requirements for the runtime environment, and in particular the sophisticated optimization mechanisms needed for developing and integrating the system components. In the meantime, a cloud environment provides virtualized, elastic, controllable and quality on demand services for supporting systems like time critical applications. However, the engineering method and software tools for developing, deploying and executing classical time critical applications have not yet included the programmability and

TABLE I: Scaling of nodes considering the data flow of the proposed architecture

Nodes Set	Provisioning	Input Flows	Output Flows
Input Distributors	N	N	N
Proxy Transcoders	$N+1$	$N+2$	$N+1$
Video Switcher	1	N	1
Outputs	M	$P+Q$	M

controllability provided by clouds; and the time critical applications cannot yet get the full potential benefits which cloud technologies could provide.

It is still an open question whether live streaming applications, like the one outlined above, are suited to run in one or more private or public cloud environments. To deploy and control such time-critical applications require a workbench of dedicated tools each having its well-defined task.

The actual nature of individual response-time constraints varies. For example, often time constraints are imposed on the acquisition, processing and publishing of real-time observations, not least in scenarios such as weather prediction or disaster early warning [8]. The ability to handle such scenarios is predicated on the time needed for customization of the runtime environment and the scheduling of workflows [9] [10], while the steering of applications during complex experiments is also temporally bounded [11]. Time constraints imposed on the scheduling and execution of tasks require high performance or high throughput computing (HPC/HTC) and depend on the customization, reservation and provisioning of suitable infrastructure, on the monitoring of runtime application and infrastructure behaviour, and on runtime controls.

III. The Architecture and Techniques of CloudsStorm Framework

The Dynamic Real-time Infrastructure Planner (DRIP) is a system developed in the SWITCH project for the planning, validation and provisioning of the virtual infrastructure to support migrating applications onto clouds. The entire SWITCH) [12] workbench also includes other two subsystems: SIDE (SWITCH Interactive Development Environment). It is responsible for composing and describing applications; ASAP (Autonomous System Adaptation Platform). It is a runtime monitoring and adaptation subsystem [13]. CloudsStorm [14] framework is the key component of DRIP we developed, simplifying the DevOps lifecycle for applications migration onto clouds. In this section, we demonstrate how the framework, CloudsStorm, works and some important techniques of the framework to mitigate the gap of challenges mentioned above.

A. Architecture Overview

Figure 2 illustrates the architecture overview of CloudsStorm framework.

In this framework, there are mainly three types of code, including the application code, infrastructure code and infrastructure description. The application code is the original logic of the application, which is going to run on clouds. The difference of our framework is that we furthermore allow the application developer to describe its underlying infrastructure and operations on the infrastructure. Among these code types, the infrastructure code is the core of the framework. Among them, the infrastructure description allows application developers to describe the infrastructure, which is required by the application. It also includes the network topology. The infrastructure code is leveraged by cloud applications to control the

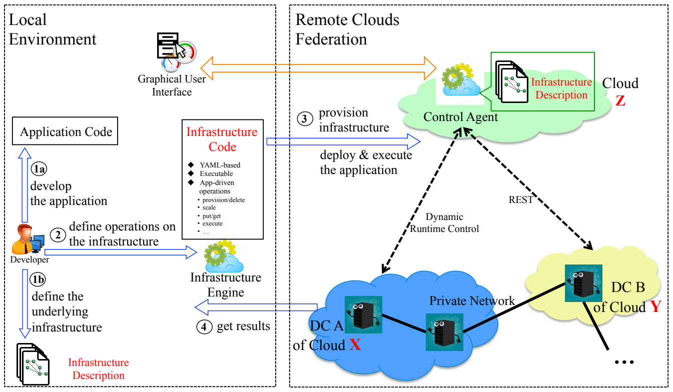


Fig. 2: Architecture overview of CloudsStorm framework

whole lifecycle of the infrastructure, including operations of provisioning, deployment, auto-scaling and destruction.

After above infrastructure definitions, the infrastructure code can be executed to perform all the operations on the infrastructure, such as provisioning, deploying and executing the application, etc. Meanwhile, the control agent is responsible for dynamically control at runtime through infrastructure code, including auto-scaling, failure recovery, etc. Finally, results generated by the application can be retrieved directly from the remote federated clouds.

B. Infrastructure management and description syntax

In CloudsStorm framework, all the related syntax is in the format of YAML (YAML Ain't a Markup Language). Because YAML format is human readable, which is easy to understand. The detailed infrastructure description is well explained in the online manual[3] or from our previous work [15].

Then we briefly introduce our partition-based infrastructure management mechanism to show the infrastructure description in our framework. Figure 3 illustrates an example topology. We classify the application-defined topology description into three levels. The lowest level is the VM level. It describes the types of VMs required, mainly referring to the computing capacities, CPU, memory, etc. The level in the middle is the sub-topology level. It includes descriptions of several VMs in one data center and also describes the cloud provider from which this data center comes from. The top level is the top-topology level. It includes all the sub-topologies and describes the network connections among these VMs. Internally, the network is defined as a private network, being that it is useful for the application to define the topology as such during the design phase. These descriptions are based on YAML format.

In this example, there are three sub-topologies from different different clouds and data centers to consist the entire infrastructure topology for hosting the cloud application. Meanwhile, there are two subnets to describe the network connection among these computing resources. Instead of using the public IP address, all the VMs can be configured to be connected with private IP addresses via CloudsStorm. These

[3]https://cloudsstorm.github.io

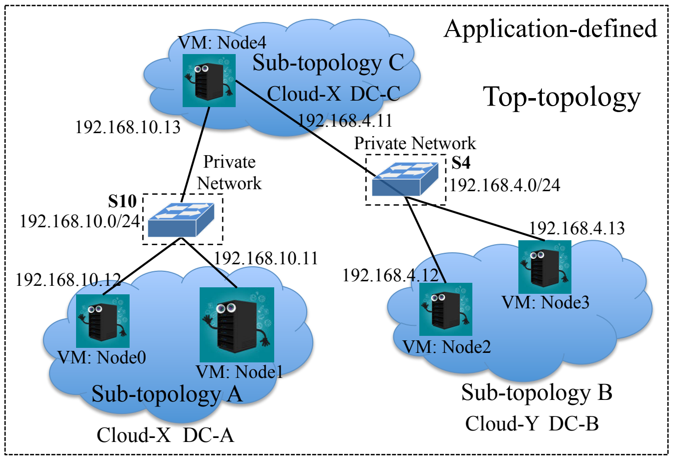


Fig. 3: An example topology description of partition-based infrastructure management

Code 1 Example code for VM level horizontal scaling

```
- CodeType: “SEQ”
  OpCode:
    Operation: “hscale”
    Options:
    - ReqID: “hs_req_1”
      CP: “ExoGENI”
      DC: “GWU (Washington DC, USA)”
      OutIn: “Out”
    ObjectType: “VM”
    Objects: “XS.Node0 || YS.Node1”
- CodeType: “SEQ”
  OpCode:
    Operation: “hscale”
    ObjectType: “REQ”
    Objects: “hs_req_1”
```

addresses are therefore able to be fixed, when the infrastructure is provisioned every time. Hence, the provisioned infrastructure according to the description is reproducible.

C. Infrastructure code and scaling operation

The infrastructure code is also organized in the YAML format. Based on the above application-defined topology description, application developers can further develop the infrastructure code to execute and run their applications on clouds. The infrastructure code is basically a set of operations defined sequentially in a list. In order to combine these basic operations to complete a complex task, we define two types of code to do the operation, ‘SEQ’ and ‘LOOP’. ‘SEQ’ code only contains one operation. A list of ‘SEQ’ codes is executed one at a time. ‘LOOP’ code contains several operations and performs repeatedly for a number of iterations. The detailed syntax definition can also be found in the online manual mentioned above.

In order to fill the gap to migrate applications onto clouds, CloudsStorm framework also provides some high-level controllability, including horizontal scaling, vertical scaling, failure recovery, etc. Code 1 below demonstrates the example of horizontal scaling at VM level (there is also corresponding sub-topology level).

In this example, the scaled part is the VM named as “Node0” from sub-topology “XS” and VM “Node1” from sub-topology “YS”. The request element shows that the scaled VM would be provisioned in the Washington data center from ExoGENI[4] cloud. The name of this scaled sub-topology is not explicitly specified. It would be named in some rules by CloudsStorm. Meanwhile, all the scaled sub-topology copies will keep the same network topology as the original scaling sub-topologies. The network IP addresses of the scaled VMs are picked from the private addresses pool of the original subnet. The symbol in ‘||’ in “objects” definition means that the scaling operation of these two VMs are performed in parallel.

Compared to other programming tools, the syntax of CloudsStorm is much more simple. For example, if adopting jclouds[5] to automate the scaling process, the developer needs advanced programming skills in Java to implement multi-thread to make the operation in parallel. And the code is hard to reconstruct to schedule these operations. On the contrary, our syntax is human readable and very easy to learn.

[4]http://www.exogeni.net/

IV. Case Study

Figure 4 demonstrates how the live streaming use case is composed by the SIDE subsystem. Four Input Distributors are connected to a Video Switcher and their respective Proxy Transcoders. From the Video Switcher, there are two outputs, one to the broadcast and another to the Output Proxy Transcoder. All the Proxy Transcoders are connected to a frontend, in order that the operator may watch the feeds. There are also REST connections between the frontend component and the Proxy Transcoders to the Video Switcher component,

[5]https://jclouds.apache.org/

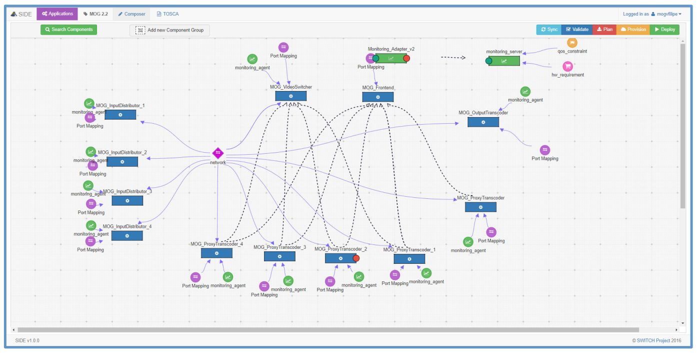
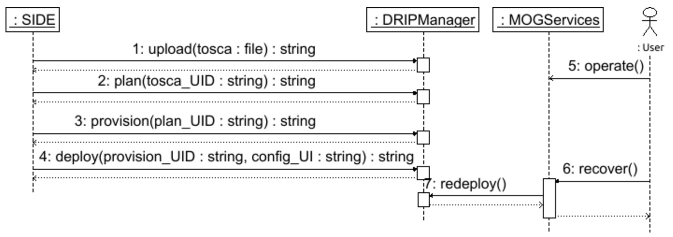


Fig. 4: The workflow of the live streaming application components



: SIDE
: DRIPManager
: MOGServices
: User
1: upload(tosca : file) : string
2: plan(tosca_UID : string) : string
3: provision(plan_UID : string) : string
4: deploy(provision_UID : string, config_UI : string) : string
5: operate()
6: recover()
7: redeploy()

Fig. 5: Sequence diagram for deploying the live streaming use case

in order to handle the operation. All these components are managed as Dockers[6].

The communication between the user and DRIP is made within SIDE. SIDE maps all the information gathered and compiles a description file, which provides an overview of the distributed components, their interdependencies and associated QoE requirements. Once completed, the description file is sent to DRIP via a POST request. Then 'compress-relax' Multi dEadline workflow Planning Algorithm (MEPA) method is leveraged to assign each task in the workflow to the best performing VM possible such that multiple deadlines are met. Wang [16] demonstrated the performance of both approaches for task graphs generated by the GGen package [17] applying the 'fan-in/fan-out' methods, showing that the MEPA method can successfully cope with these kinds of problems and allows for an easy adaptation in case more constraints play a role. Therefore, the user only need to do some drag and drop configurations, and later fill the remaining data in the components. Because of our CloudsStorm framework, the developer does not need to know any special language or create a document consisting of thousands of lines of code, in order to cooperate with the cloud infrastructure.

The plan given by DRIP is according to collected performance information. The systematic collection and sharing of such information will allow the DRIP to select the most suitable resources for mission-critical applications. Elzinga [18] showed the functionality of this collector.

[6]https://www.docker.com/

Code 2 A basic top-topology description for the live streaming application

```
topologies:
- topology: MOG
  cloudProvider: EC2
  domain: California
  status: fresh
subnets:
- name: s1
  subnet: 192.168.10.0
  netmask: 24
  members:
  - vmName: MOG.nodeA
    address: 192.168.10.10
  - vmName: MOG.nodeB
    address: 192.168.10.11
```

Code 3 A basic sub-topology description for the live streaming application

```
VMs:
- name: nodeA
  nodeType: t2.medium
  OSType: Ubuntu 16.04
  script: applicationInstall.sh
  role: master
  publicAddress: null
- name: nodeB
  nodeType: t2.medium
  OSType: Ubuntu 16.04
  script: applicationInstall.sh
  role: slave
  publicAddress: null
```

When DRIP receives the description file of components, it is saved under the user's account with a unique identifier. The CloudsStorm then is invoked to materialize the virtual infrastructure along with the necessary cloud credentials stored in the manager to request resources from one or more cloud providers. It is able to provision a networked infrastructure, recover from sudden failures quickly, and scale across data centers or clouds automatically [19]. This framework is able to set up a networked virtual Cloud across even public clouds, which do not explicitly support network topology, like EC2[7] or EGI[8]. Finally, the deployment agent of CloudsStorm uses the description to arrange application components in the virtual infrastructure. Figure 5 illustrates the process to migrate this use case form MOG company.

Code 2 and 3 demonstrate a basic topology description leveraged for the underlying virtual infrastructure to run the live streaming application. Code 2 is a top-topology level description. It shows that there is only one sub-topology, named as 'MOG', for this application. It is planned from EC2 cloud and the data center at California. Meanwhile, the network connection is also defined in this file, where the

[7]https://aws.amazon.com/ec2/
[8]https://www.egi.eu/federation/egi-federated-cloud/

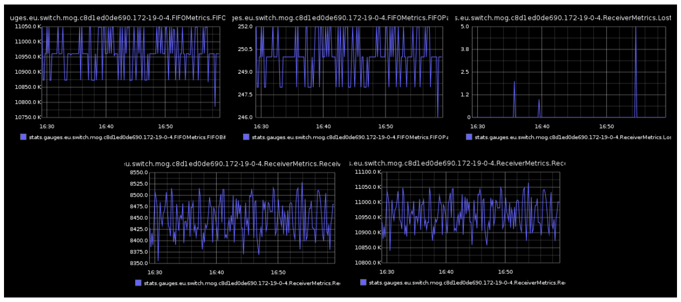


Fig. 6: Application level metrics sent by output transcoders

two VMs are defined in a private subnet. Code 3 is the detailed sub-topology description of 'MOG'. It contains the required two VMs, including OS type, computing capacity, etc. The "script" refers to the script file specifying how the computing environment and the application would be installed. The "publicAddress" field equals "null" in this design phase, because the VM has not actually been provisioned. In this case, these two VMs are going to be managed by Kubernetes[9] as a Docker cluster. Afterwards, the application, which consists of Dockers, is deployed.

Finally, CloudsStorm provides a deadline-aware deployment scheduling for time-critical applications in clouds comes into action, which accounts for deadlines on the actual deployment time of application components [20]. This is of special importance for horizontal scaling afterwards.

After those steps, the application can be in operation on clouds for live steaming. The application level metrics are retrieved from the final output transcoders as shown in Figure 6. It shows that the QoE requirement of this application is satisfied. This information is also useful to decide whether to automatically adapt the infrastructure to meet the QoE requirements, when the performance is low.

V. Conclusion

In this paper, we discussed the DevOps challenges for migrating the live streaming application from a legacy physical system to clouds, and present a framework called CloudsStorm to automate the procedure for planning, provisioning and deploying live steaming applications based on their time constraints. In the paper, the time critical constraints are not only referring to the as fast as possible but also to the deadlines that application has to meet.

The use of cloud for services that are typically performed by specialized physical hardware, such as video switching or transcoding, can be virtualized for on-demand pay-per-use services. But the advantages of the cloud are not just related to the business model. The ability to easily build an infrastructure anywhere in the world without the need to spend days configuring the entire infrastructure is enough to be attractive in and of itself.

The CloudsStorm, embedded in SWITCH workbench, supports the building of a distributed application in the cloud for production of events using remote production studios. Not only it is the feasibility of the architecture demonstrated, but also the feasibility of the various possible expansion possibilities, making it the beginning of a possible solution to be worked on and improved in the future.

Acknowledgment

This research is funded by the EU Horizon 2020 research and innovation program under grant agreements 643963 (SWITCH project), 654182 (ENVRIPLUS project) and 676247 (VRE4EIC project).

[9]https://kubernetes.io/

References

[1] D. ETSI, "Transport of mpeg-2 ts based dvb services over ip based networks," *ETSI TS*, vol. 102, no. 034, p. V1.

[2] A. Kovalick, "The fundamentals of the all-it media facility," *SMPTE Motion Imaging Journal*, vol. 123, no. 2, pp. 24–30, 2014.

[3] L. Chiariglione, "Mpeg-2-generic coding of moving pictures and associated audio information," *ISO/IEC JTC1/SC29/WG11*, 2000.

[4] D. Hoffman, G. Fernando, V. Goyal, and M. Civanlar, "Rtp payload format for mpeg1/mpeg2 video," Tech. Rep., 1997.

[5] K. Jeferry, G. Kousiouris, D. Kyriazis, J. Altmann, A. Ciuffoletti, I. Maglogiannis, P. Nesi, B. Suzic, and Z. Zhao, "Challenges emerging from future cloud application scenarios," *Procedia Computer Science*, vol. 68, pp. 227 – 237, 2015, 1st International Conference on Cloud Forward: From Distributed to Complete Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050915030835

[6] Z. Zhao, P. Martin, J. Wang, A. Taal, A. Jones, I. Taylor, V. Stankovski, I. G. Vega, G. Suciu, A. Ulisses *et al.*, "Developing and operating time critical applications in clouds: The state of the art and the switch approach," *Procedia Computer Science*, vol. 68, pp. 17–28, 2015.

[7] T. Kojima, J. J. Stone, J.-R. Chen, and P. N. Gardiner, "A practical approach to ip live production," *SMPTE Motion Imaging Journal*, vol. 124, no. 2, pp. 29–40, 2015.

[8] S. Poslad, S. E. Middleton, F. Chaves, R. Tao, O. Necmioglu, and U. Bügel, "A semantic iot early warning system for natural environment crisis management," *IEEE Transactions on Emerging Topics in Computing*, vol. 3, no. 2, pp. 246–257, 2015.

[9] Z. Zhao, P. Grosso, J. Van der Ham, R. Koning, and C. De Laat, "An agent based network resource planner for workflow applications," *Multiagent and Grid Systems*, vol. 7, no. 6, pp. 187–202, 2011.

[10] Z. Zhao, D. Van Albada, and P. Sloot, "Agent-based flow control for hla components," *Simulation*, vol. 81, no. 7, pp. 487–501, 2005.

[11] K. Evans, A. Jones, A. Preece, F. Quevedo, D. Rogers, I. Spasić, I. Taylor, V. Stankovski, S. Taherizadeh, J. Trnkoczy *et al.*, "Dynamically reconfigurable workflows for time-critical applications," in *Proceedings of the 10th Workshop on Workflows in Support of Large-Scale Science*. ACM, 2015, p. 7.

[12] Z. Zhao, A. Taal, A. Jones, I. Taylor, V. Stankovski, I. G. Vega, F. J. Hidalgo, G. Suciu, A. Ulisses, P. Ferreira *et al.*, "A software workbench for interactive, time critical and highly self-adaptive cloud applications (switch)," in *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*. IEEE, 2015, pp. 1181–1184.

[13] S. Taherizadeh, A. C. Jones, I. Taylor, Z. Zhao, and V. Stankovski, "Monitoring self-adaptive applications within edge computing frameworks: A state-of-the-art review," *Journal of Systems and Software*, vol. 136, pp. 19 – 38, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016412121730256X

[14] H. Zhou, Y. Hu, J. Su, C. de Laat, and Z. Zhao, "Cloudsstorm: An application-driven framework to enhance the programmability and controllability of cloud virtual infrastructures," in *International Conference on Cloud Computing*. Springer, 2018, pp. 265–280.

[15] H. Zhou, Y. Hu, C. de Laat, and Z. Zhao, "Empowering dynamic task-based applications with agile virtual infrastructure programmability," in *IEEE International Conference on Cloud Computing*. IEEE, 2018.

[16] J. Wang, A. Taal, P. Martin, Y. Hu, H. Zhou, J. Pang, C. de Laat, and Z. Zhao, "Planning virtual infrastructures for time critical applications with multiple deadline constraints," *FGCS*, 2017.

[17] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner, "Random graph generation for scheduling simulations," in *Proceedings of the 3rd international ICST conference on simulation tools and techniques*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2010, p. 60.

[18] O. Elzinga, S. Koulouzis, A. Taal, J. Wang, Y. Hu, H. Zhou, P. Martin, C. de Laat, and Z. Zhao, "Automatic collector for dynamic cloud performance information," in *Networking, Architecture, and Storage (NAS), 2017 International Conference on*. IEEE, 2017, pp. 1–6.

[19] H. Zhou, J. Wang, Y. Hu, J. Su, P. Martin, C. De Laat, and Z. Zhao, "Fast resource co-provisioning for time critical applications based on networked infrastructures," in *Cloud Computing (CLOUD), IEEE International Conference on*, 2016, pp. 802–805.

[20] Y. Hu, J. Wang, H. Zhou, P. Martin, A. Taal, C. de Laat, and Z. Zhao, "Deadline-aware deployment for time critical applications in clouds," in *European Conference on Parallel Processing*. Springer, 2017, pp. 345–357.