Journal of Network and Systems Management (2020) 28:1391–1426
https://doi.org/10.1007/s10922-020-09543-y

Preemptive Resource Provisioning for Container-Based Audio/Video Encrypted Collaboration Applications

Rafael Xavier[1] · Lisandro Zambenedetti Granville[2] · Filip De Turck[1] · Bruno Volckaert[1]

Received: 14 August 2019 / Revised: 19 May 2020 / Accepted: 26 May 2020 / Published online: 11 June 2020
© Springer Science+Business Media, LLC, part of Springer Nature 2020

Abstract
The massive industrial adoption of cloud technology has led to research into cloud-enabling traditional applications. The EMD research project proposes an elastic, reliable, and secure cloud-enabled Audio and Video (A/V) collaboration platform in replacement of a reliable hardware appliance based which had fixed constraints in terms of scalability. In this context, this article introduces heuristics and architectures that efficiently and preemptively allocate EMD's A/V encrypted and container-based software components in the cloud. A software solution based on Kubernetes, a production-grade container orchestration platform, is compared with another solution focused on dedicated VMs. Both implement resource allocation heuristics that take into account the project's requirements and location-aware encryption enforcement necessities: encryption is enforced for more sensitive data. A company training scenario with dynamically distributed instructors is modelled using existing A/V stream concepts, and component prototypes are extended to support encryption and containerisation, whose prototype performance evaluation drives the investigation of heuristics and architectures and feeds their larger-scale simulation-based assessment. Results show that container orchestration costs are at least 52% lower than dedicated VMs for this scenario, but rely on relaxing a project requirement: the time taken to establish a new streaming session was to be kept below 2 s. The switch to orchestrated containers raised this up to a maximum of 2.5 s.

Keywords Cloud · Audio/Video · Streaming · Container orchestration

✉ Rafael Xavier
rafael.xavier@ugent.be

Extended author information available on the last page of the article

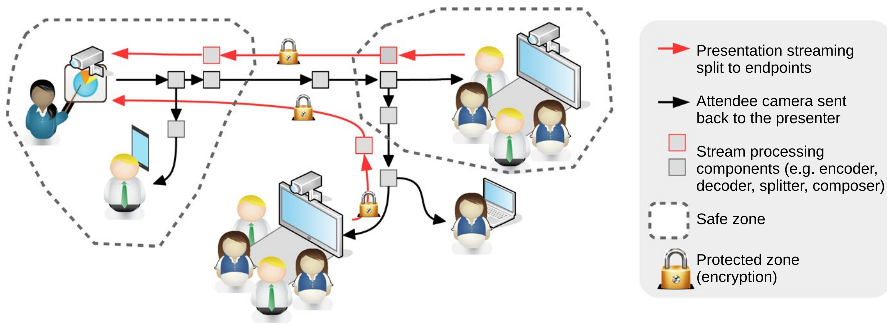

Fig. 1 Example of collaboration scenario studied in this article: one instructor sends his A/V streaming data to multiple attendees, which may return their camera's images to the instructor. Users are positioned inside or outside pre-defined safe zones, and platform components process the data in order to encrypt it or adapt it to user endpoint's capabilities

1 Introduction

Cloud hosting models offer advantages such as flexibility and global distribution, with the ability to rapidly scale to massive amounts of users. The maturity of this model brings research efforts focused on the conversion of traditional solutions to this new paradigm. One example of collaboration handled by the platform is depicted in Fig. 1. Multiple bi-directional streaming flows carry course presentation A/V data in a one-to-many fashion, from an instructor endpoint to a select group of attendees' endpoints (flows are represented by black arrows). Attendees' camera images are also sent to the instructor (red arrows), enabling interaction.

Endpoints with distinct stream processing capabilities can join the session (e.g.,: one endpoint can have the processing capacity to source/receive up to 720 p encrypted A/V streams, others may handle up to 480 p or 1080 p without encryption). Public and private networks are utilised, raising additional security risks, and security zones (safe zones, unsafe zones) are defined to drive location-aware encryption policy decisions. Additionally, too long waiting times can harm the user experience (i.e., the delay between attempting to establish a connection and seeing the first A/V content). The EMD research project, through discussions held with industry partners that developed the hardware-based solution, proposed a deadline of 2 s for joining a software-based A/V collaboration.

Supporting the collaboration session, platform software components (represented by the squares in Fig. 1) are allocated in distributed data centres and process the A/V data, adapting it to endpoint's capabilities and platform needs (e.g., content delivery in time; distribution of video sources in a one-to-many fashion; transformation of A/V data's compression standards, codecs and resolutions; encryption of data streams).

Project requirements include heuristics for efficient resource allocation under a controlled budget (with a flexible budget that must be kept in check to guarantee viability of the solution). Previous work analysed similar scenario and requirements, with the use of A/V collaboration in an educational setting, and proposed

a stream model and budget-oriented resource allocation algorithms [1]. The collaboration was supported by platform components dynamically configured and deployed in VMs. However, encryption policies were not enforced. This initial research also focused on dedicated VMs when hosting platform components (i.e., one software component per VM), which required a high number of VMs to fit the varied A/V component set it utilised, significantly increasing VM costs.

In this article, we propose a streaming model and resource allocation heuristics that enforce encryption policies and intelligently and preemptively allocate orchestrated container-based platform components over distributed cloud data centres, under strict project requirements, while keeping budget in check. Two solutions are investigated: one based on dedicated VMs, with one A/V streaming component running per VM; and another based on containers orchestrated by Kubernetes [2], a production-grade container orchestration solution. In our solution, Kubernetes nodes run on top of VMs, resulting in multiple containerised A/V components running per node, and consequently per VM. While the solutions that dedicate VMs utilises a varied set of predefined VMs, the smaller set of VM necessary when orchestrating containers improves resource usage and costs. They are evaluated using an extended version of our CloudSim simulation framework. To support this research and the proposed simulation-based evaluation methodology, streaming software components are prototyped and ported to run in Docker [3] containers, and evaluated using the Virtual Wall [4] test bed. Results provide statistics w.r.t. resource usage, processing delay, and containerisation and orchestration overhead, feeding our simulation framework with a more realistic dataset.

This article is structured as follows. First, related work is analysed in Sect. 2. Then, the streaming model and project requirements are presented (in Sect. 3). Section 4 details the proposed software architectures. The heuristics that intelligently allocate platform components are listed in Sect. 5. Sections 6 and 7 describe the results obtained from component prototyping and large scale scenario simulations, respectively. Finally, Sect. 8 concludes the article by presenting our final considerations and potential future research avenues.

2 Related Work

The studied scenario comprises collaboration sessions with instructors and attendees dynamically and geographically placed in different locations. Collaboration data flows in a one-to-many fashion and platform encryption-enabled components are distributed among the cloud data centres to support them. Component placement allocation heuristics are proposed to decide where software needs to be allocated among distributed cloud data centres, and these heuristics are implemented by two proposed software solutions, one based on dedicated VMs and another that orchestrates containers. Such solutions, described in the next sections, are proposed based on a data set obtained from component prototyping, which seeds our simulation framework when evaluating larger collaboration scenarios.

2.1 Location-Aware Security Policies

Cloud computing and mobility brings security concerns depending on where data is transmitted (e.g. external or public infrastructures). Several security incidents reinforce these concerns [5–9]. To counter this problem, several studies take location into account when proposing security solutions. Encryption and access-control services were proposed to tackle location-based security decisions [10–12]. However, they do not tackle the placement of security-enabled streaming components based on A/V data location and specific associated risks. In this article, we propose the enforcement of end-to-end encryption mechanisms in the application layer based on where (in the network) the data flows. Other analysis and mitigation approaches can be valid when targeting risk reduction. Researchers aim to protect user data by enforcing encryption on web services and streaming [13, 14] or to detect new vulnerabilities and improve protection mechanisms [15]. This article does not focus on advancing the state-of-the-art in secure data transmission but on the intelligent placement of components that use state-of-the-art encryption mechanisms in the cloud.

2.2 Cloud Resource Allocation

Many efforts have focused on cloud resource allocation [16–20]. However, they did not take into account the location of endpoints. This article studies endpoints placed outside the cloud data centres and connected by public/private networks (e.g., mobile or home users), and the way they reach the platform components allocated in the data centres is a primary issue.

Another important point is the placement of the collaboration platform's components. Network Functions Virtualization (NFV) [21] offers complex network services by allocating, configuring, and chaining network functions [22, 23]. However, NFV-based management systems focus on generic network functions for a single federation. Our approach focuses on A/V specialised components composing workflows and crossing many federations (e.g., a stream originated by a presenter in one location can be sequentially processed in two additional locations (data centres A and B), and then be delivered to an endpoint in a fourth location).

With low-overhead containers advancing on the replacement of VMs in many cases, orchestration platforms need intelligent heuristics to distribute them among their work nodes efficiently. Kubernetes implement heuristics to allocate pods under strict node capacity checks [24]. However, this article proposes the pre-allocation of stand-by pods, with requirements not yet configured at the time of pod creation and replication (container allocation algorithms configure them after choosing to host components). When allocating a platform component, the proposed container allocation algorithms implement bin-packing [25] heuristics to efficiently organise and choose pre-loaded pods.

2.3 Cloud-Based A/V Services Allocation

Adaptive and Dynamic Scaling Mechanism (ADS) is a scalable and elastic solution that targets smaller scenarios (i.e., distance learning) to large ones (i.e., Massively Multiplayer Online Game (MMOG) platforms) [26]. Optimised media service scheduling algorithms were proposed [27] to provide acceptable user experience under low waiting times. However, these efforts do not handle specialised A/V components. Approaches focused on the EMD research project's professional A/V collaboration needs have been proposed, with similar scenario, components, models and concepts [28], preemptive VM management techniques [29], and more complex component workflows [1, 30, 31]. However, these efforts did not consider data protection or containerisation.

2.4 Security-Enabled A/V Streaming Components

This article includes encryption to understand the resource usage impact for professional A/V collaboration solutions. Several options are available for streaming encryption. Considering Real Time Protocol (RTP) streaming sessions, some approaches can be adopted as protection: RTP over Real Time Protocol (TLS), RTP over Datagram Transport Layer Security (DTLS), Secure Real-time Transport Protocol over Datagram Transport Layer Security (DTLS-SRTP), Internet Protocol security (IPSEC) tunnels, among others [32]. WebRTC [13] is a widely adopted standard, an architecture that enables secure A/V conferencing over the Internet, and implements DTLS-SRTP to protect the data between pairs of endpoints when necessary. However, WebRTC is initially designed for one-to-one communication, and more efficient one-to-many features must be built to reach multiple users and avoid complex/costly mesh networks [33]. This article focus on a A/V collaboration platform for multiple endpoints, streaming UDP data in a one-to-many fashion. Therefore, we focus on DTLS-SRTP to build secure A/V streaming components and provide a flexible model integrating these necessities.

3 A/V Collaboration Requirements and Concepts

This section presents *A/V* stream concepts, platform software components, and project requirements used to model software architectures and service-to-resource provisioning heuristics proposed in this article.

3.1 EMD Project Requirements

The first requirement is the time required to start a new streaming transmission (or the time for a user to join an existing collaboration session). A long waiting time can jeopardise the user experience, and initially we parameterized this as a 2 s deadline.

A second requirement is the secure transmission of sensitive data, as the establishment of remote A/V collaboration sessions over public/private networks raises security risks. We propose the enforcement of location-aware encryption policies depending on how critical the transmitted content is.

Finally, the platform is designed to run on Infrastructure-as-a-Service clouds (IaaS), with software components hosted by dedicated VMs or by containers running on them. Therefore, a software solution is necessary, and A/V streaming flows and A/V software must be modelled. Additionally, imposed cloud costs are critical to the economic viability of the proposed platform and must be kept in check, and intelligent heuristics must manage resources efficiently.

3.2 Stream Data Concepts

Video codecs employ compression/decompression algorithms that implement video standards e.g., h.264 AVC, h.265 HVEC and raw (for non-encoded data). *Data format* is the combination of a codec, an encryption function and a video resolution [e.g., 1920 × 1080 (1080 p), 1280 × 720 (720p)]. The encryption function represents the encryption algorithms applied after encoding the content [e.g., AES, 3DES, plain (absence of encryption)].

The A/V *content* is standardised and prepared by codecs and data formats to be transmitted and displayed. The content, when combined with a data format, composes a *stream data*. Therefore, K_d is the stream data representation of a sourced content K using the data format d. Additionally, multiple endpoints can dynamically join or leave the streaming session and request the same content. While the content is produced by only one endpoint, it can be distributed in a one-to-many fashion by dynamically allocated specialised A/V components, and delivered to multiple endpoints. As multiple stream sessions can occur in parallel, the set of interconnected A/V components that process or transform all content handled by the platform is called *stream workflow*, a dynamic structure that changes on demand. The components that compose stream workflows are described next.

3.3 Collaborative Streaming Workflow Components

Platform software components have specific functionalities. They receive a set of stream data flows as input, process them according to their functional intent, and provide a resulting stream data set as output. A component is represented as $C_{I,O,\alpha}$, being I the input data stream set, O the output data stream set and α the internal processing function. For instance, transcoding is the conversion of content from one data format to another. Therefore, a resolution transcoder $T_{\{C_{1080p}\},\{C_{720p}\},\beta}$ receives content C encoded in 1080 p and converts it to 720 p by using the conversion function β. Figure 2 depicts the stream components used in this article, which are described below.

Five components are defined: *tee*, *encoder*, *decoder*, *transcoder* and *compositor*. Encoders convert raw streaming data formats into compressed/encrypted formats, while decoders convert compressed/encrypted data formats to raw. Transcoders

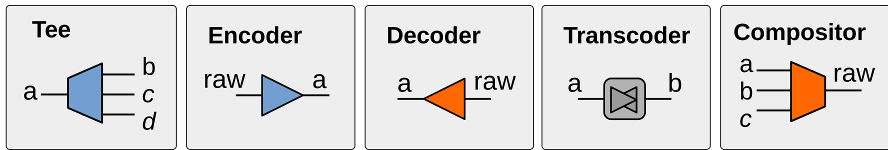

Fig. 2 Components used to model A/V stream workflows. Tees, encoders, decoders, transcoders and compositors are depicted with examples of inputs/outputs

implement the transformation between two data formats by changing codec, resolution, or encryption mode. A tee receives a stream data flow and distributes it in a one-to-many fashion. Tees are internally composed of a core A/V video splitter (able to receive raw data and replicate it to multiple outputs) and multiple transcoders attached to input and outputs. These transcoders enable the possibility of distinct data formats for input and outputs. Otherwise, a tee would handle only one data format. Finally, compositors receive multiple inputs, using distinct data formats, and generate new video-in-video content based on the received inputs. They are internally composed of multiple decoders (one per input) and a core mixer, which processes the video-in-video composition.

We are investigating a solution for large dynamic A/V collaborations, where it can be beneficial to show the streams of multiple participants at once. Highly configurable endpoints are expected to flexibly compose video-in-video (sometimes also called picture-in-picture) outputs under user control. For instance, the endpoints can display multiple combined inputs on multiple screens, and the inputs can be resized, omitted, and overlapped. This flexible video-in-video composition is performed by compositor's internal mixers. Considering that client endpoints are not allocated using cloud resources (instead, they run on the client machine), and the complexity of designing these flexible endpoints, this article focus on the component chains up to the decoders, leaving the mixer/endpoint design out of scope.

3.4 Stream Workflow Composition

When starting new A/V data transmissions, components are instantiated and interconnected by the platform, forming component chains. To connect two components X and Y, an output from X must be connected to an input of Y, and both must utilise

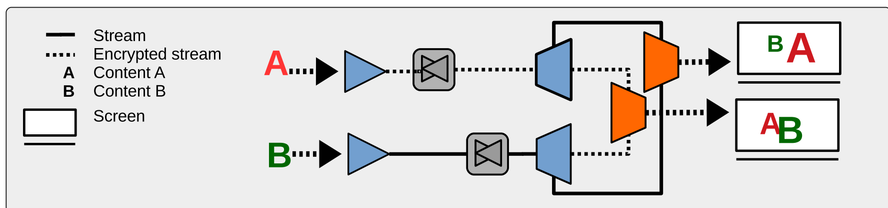

Fig. 3 Two distinct A/V sources A and B are streamed, being split by tees and transmitted over different paths. Transcoders process the data on the way, while decoders and compositors prepare video-in-video outputs to be displayed on screens

the same data format. Figure 3 depicts a workflow example. Encoders, decoders, tees, compositors, and transcoders are chained to transmit two contents (A and B) and display them on two screens.

4 Design of Container-Based Media Collaboration Architecture

This section describes the research on a software solution able to manage the required cloud infrastructure, with modular resource allocation heuristics deciding on component-to-resource provisioning of the collaboration platform's A/V streaming components.

4.1 Orchestration Model Investigation

Three distinct scenarios were investigated, prototyped, and evaluated on support of architecture design decisions: *dedicated VMs*, with A/V software components running in dedicated VMs; *unorchestrated containers*, with A/V software components running in non-orchestrated containers; and *orchestrated containers*, with A/V software components running in containers managed by a Kubernetes container orchestration platform.

To have a proper understanding of the behaviour of the components in these three scenarios, streaming software components were prototyped, ported to run in Docker containers, converted into Kubernetes workloads, and evaluated using a set of VMs. The complete prototyping evaluation is reported in Sect. 6. In this section, we describe a result relevant for the solution investigation effort: the time taken to request a new A/V component, from the generation of a request to the moment it is up and running in the system, as shown in Table 1.

The measured delays are part of the stream establishment time, as A/V components must be requested, configured, and interconnected when forming stream workflows. The results show that dedicated VMs are a low instantiation delay option (with no containerisation or orchestration overhead). However, unorchestrated containers impose delays incompatible with the 2 s deadline because of a high overhead when loading Docker containers (in terms of time to deploy new containers). On the other hand, the use of orchestrated containers provided lower instantiation times, a result of a technique that uses Kubernetes features to pre-deploy stand-by containers. The incompatibility of unorchestrated containers with the requirements

Table 1 Time taken from the request of a new A/V component to the moment it is loaded and ready to be utilised

Component	Encoder (ms)	Decoder (ms)	Tee (ms)	Transcoder (ms)
Dedicated VMs	111	110	212	210
Orchestrated containers	1230	1235	1343	1347
Unorchestrated containers	2520	2486	3963	3935

described in Sect. 3 motivated us to investigate only two architectures, one based on dedicated VMs and another that orchestrates containers using Kubernetes.

4.2 A/V Collaboration Architecture Based on Dedicated VM Handling

The architecture, depicted in Fig. 4, uses no containerisation or container orchestration. It targets the allocation of one platform component per VM, and defines three layers.

Iaas is the cloud hosting service. It provides VMs when instructed by VM request API calls. *VM pool* implements VM management algorithms, which receives VM requests, keeps a pool of stand-by VMs based on demand prediction techniques, and requests new VMs from the IaaS layer when necessary to scale the solution. Finally, the *component layer* implements component allocation algorithms. This layer handles stream requests, deciding which components to allocate and where they must be placed, and requesting VMs in the appropriate locations (distributed data centres) from the *VM pool*.

When a VM component is being allocated, the VM pool looks up the template that best fits the component's requirements in terms of CPU and memory. As the requirements of components vary with the complexity of its internal features and of the streaming data they handle, a set of VM templates must be planned to fit the necessary software components and reduce resource wasting (e.g., by hosting small components in over-dimensioned VMs).

4.3 A/V Collaboration Architecture Based on Orchestrated Containers

The scenario handled by our architecture is exemplified in Fig. 5. Multiple collaboration sessions are hosted on the platform (a green, a yellow, and a red one). Each session requires software components to distribute and process the A/V data (red, yellow, and green orbs). These components are allocated on shared resources, VMs that are configured as Kubernetes worker nodes, and are located in distributed data centres.

The architecture is depicted in Fig. 6. Compared to the solution focused on dedicated VMs, one more layer is added: the *orchestration layer*. It implements a

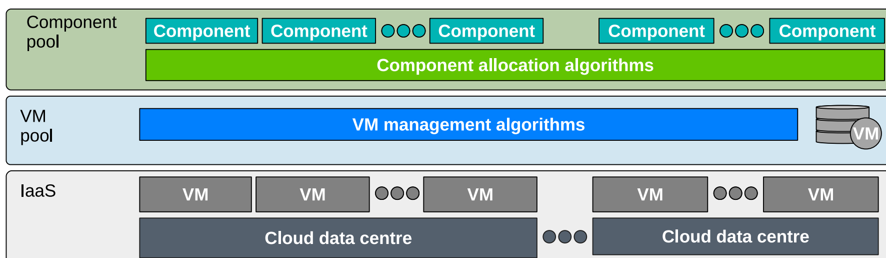

Fig. 4 Architecture proposed to allocate platform components without containerisation or container-orchestration. Three layers are presented, *component layer*, *VM pool* and *iaas*

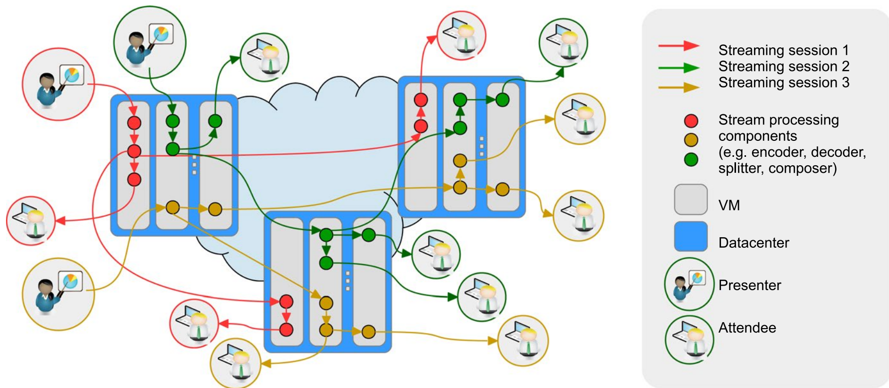

Fig. 5 Scenario handled by the *orchestrated containers* architecture. Data centres are depicted in blue, and multiple concurrent collaboration sessions are supported by A/V components allocated on multiple VMs (Kubernetes worker nodes)

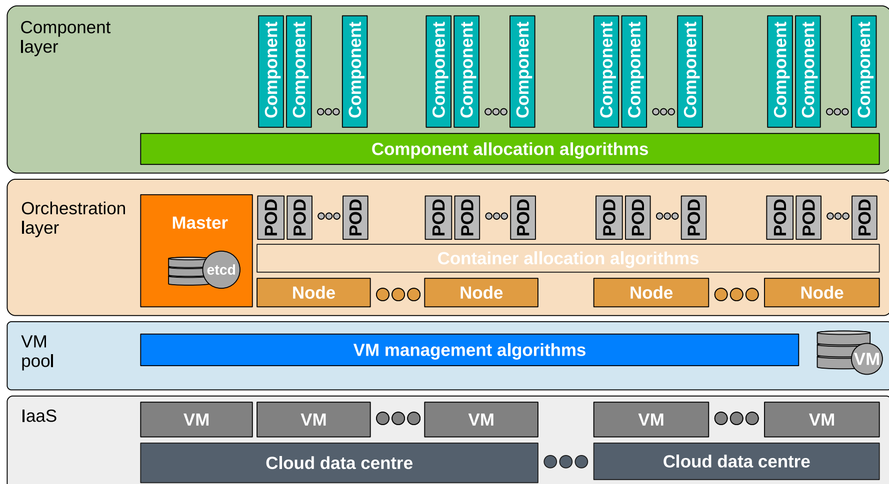

Fig. 6 The architecture based on orchestrated containers implements four layers: *IaaS*, providing VMs; a *VM pool* layer, managing a preemptively populated pool of available VMs; an *orchestration layer*, that implements a Kubernetes container orchestration solution; and a *component layer*, that decides how components will be chained and provisioned to pods to support incoming stream requests

Kubernetes container orchestration solution, which has two main node types: master nodes, which control the cluster based on a configuration database (etcd) and API calls; and worker nodes, the work units that process workloads. Kubernetes workloads are called pods, in our case composed of one Docker container hosting A/V streaming components. This layer also implements container allocation algorithms that optimise the placement of containers in the available nodes (bin-packing algorithms). New VMs are requested from the VM pool when necessary to deploy more Kubernetes worker nodes and scale the solution. Finally, the *component layer*

implements component allocation algorithms, which receives stream requests, decides which components to allocate and where they must be placed. Pods are provided by the *orchestration layer* in the appropriate locations (distributed data centres) to host allocation candidate components.

Our pods are composed of one Docker container that encapsulates the A/V components depicted in Fig. 2. As shown in Table 1, the loading of Docker containers results in excessive overhead when using non-orchestrated containers, jeopardising platform requirements listed in Sect. 3.1. To counter this behaviour, a pod predeployment feature was designed to be utilised when orchestrating containers. The new feature was implemented using the Kubernetes pod replication feature, which allows pod pre-loading (i.e., a pod can be loaded and configured to be replicated at a predefined number of times, and every time one pod is finished a new one is automatically loaded). The replication feature loads these pods beforehand, and they remain in stand-by mode waiting to be selected by the component allocation algorithm to host new components. When loaded, a pod tailored to host the A/V component that needs to be allocated is chosen and receives additional configuration and interconnection parameters (e.g., address of the next component to connect to, data transmission communication settings, data format). When using the solutions based on dedicated VMs, the *VM pool* manages VMs based on a pre-defined set of VM templates. However, VMs are used to host Kubernetes worker nodes instead of directly A/V components. Pods are selected to host components. If no pod is available, the VM pool provides a new VM to host a new Kubernetes worker node, scaling the solution and then re-attempting the allocation. Fewer template variations are necessary, and the bin-packing algorithms can increase the component-to-resource allocation efficiency by spawning multiple components on the same Kubernetes worker node. The algorithms that handle component allocation, and VM and container pre-allocation, are described in details in the next section.

5 Intelligent Resource Allocation

Components must be allocated taking into account the requirements detailed in Sect. 3, and must implement resource allocation functionalities required by the proposed solution.

5.1 VM Management Algorithms

The EMD research project proposes a 2 s deadline in establishing new streaming flows. Considering that the cold start of a new VM can last several minutes [34], VM provisioning must be performed beforehand to support demand in time. The algorithms described in this subsection were proposed in previous research [1] and implement the preemptive VM provisioning modelled in the VM pool layers. Each data centre has one VM pool, a structure designed to manage all available VMs, which implement the VM management algorithms. The proposed heuristics keep the VM pool populated by preemptively instantiating new VMs based on demand

prediction. Therefore, if a VM is requested and it is available, it is immediately delivered. Otherwise, the instantiation process of a new VM is started, holding the request until the VM is up and running and ready to be delivered.

Three algorithms were proposed: request rate-based algorithm (RATE), history-based algorithm (HISTORY), and the combination of both, named RATE+HISTORY. RATE detects VM request rate increases per VM template, and provides VMs at a higher rate in an attempt to supply future demands. HISTORY utilises a different approach: it analyses a time-windowed VM request history, looking back a parametrisable number of days to predict upcoming behaviour.

RATE behaves well for intense VM usage, but not without demand (platform's idle moments). It relies on a parametrisable minimum amount of operational VMs kept in standby to respond to initial demands. These resources remain active even when no demand is detected, causing an undesirable operation cost raise. HISTORY can deal with preexisting seasonal meeting patterns but shows a predisposition to bad performance when dealing with single non-recurring requests. Therefore, the combination of RATE and HISTORY is proposed: RATE+HISTORY. It relies on historical data and also responds to sudden increases in demand, requiring less stand-by VMs and keeping higher efficiency levels with lower operation costs. Therefore, in this article we employ the RATE+HISTORY algorithm for both proposed software solutions, the one that dedicates VMs and the one that orchestrates containers (in the latter case, the VMs constitute the resource pool where Kubernetes is run on).

5.2 Component-to-Resource Provisioning Algorithms

In a budget-aware manner, and based on security policies and project requirements, these algorithms decide on which data centre components are allocated, and in which order they are interconnected to form stream workflows.

The investigated scenario defines safe and unsafe zones, as shown in Fig. 1. Unsafe zones tend to be less protected and prone to higher risk levels than safe zones. We define location-aware security policies based on a risk assessment that utilises the ISO 27005 standard [35]. It defines risk as the potential of having a threat harming the organisation by exploiting vulnerabilities of assets. Therefore, the risk is measured combining the likelihood and the impact of an incident (e.g., the costs associated with a likely image or financial loss). Following, we correlate the impact with the sensitivity of the information. Sensitivity is classified as *confidential*, for confidential data (e.g., new projects, data under non-disclosure agreements, data that must be possessed only by authorised people), *internal* for internal training courses, non-critical reports, data that must stay inside the company boundaries, or *public*, for public statements, advertising projects, and other information that can be made public. For the threat likelihood, we consider that unsafe zones have fewer security controls available, and attempts to compromise the system are expected to occur more often (e.g., endpoints placed on the Internet are more prone to viruses, external networks are more susceptible to man-in-the-middle attacks).

Two unacceptable risks were identified: unprotected internal information transiting through unsafe zones, while they should transit only internally, and unprotected confidential information in transit, while they must always be protected. For these cases, the chosen security mitigation technique is encryption, which can be enforced by security policies. We mapped this necessity into three proposed policies:

- *No encryption*, for when no unacceptable risks are detected (for public information or internal information transiting through internal networks);
- *Partial encryption*, encrypting data only when using external links/Internet (unsafe zones), a policy that targets internal information in transit externally;
- *Total encryption*, which is aimed at confidential information, which must always be encrypted.

These policies are mapped into component allocation algorithms, which allow performing the policy-based establishment of encrypted/non-encrypted streaming sessions. Two auxiliary functions are defined in support of the proposed component allocation algorithms, *link*() and *linkCrypto*(). *link*(C) chains a set of components in the order that they are received. For an input $C \leftarrow \{C_1, C_2, C_3\}$, it will interconnect C_1 to C_2 and C_2 to C_3, creating a $C_1 \rightarrow C_2 \rightarrow C_3$ chain. It also introduces transcoders when data formats are not compatible. For instance, for an input $C \leftarrow \{C_1, C_2\}$, with C_1 providing data format x and C_2 expecting data format y, a transcoder that converts x to y is included, producing a $C_1 \rightarrow \textit{transcoder} \rightarrow C_2$ chain. Additionally, transcoders' placement decisions are based on bandwidth consumption. For instance, if a data format x requires more bandwidth than the data format y, the transcoder will be allocated in the same data centre as C_1; otherwise, at C_2's data centre. This measure guarantees that the data format that requires less bandwidth will be adopted when sending data between distinct data centres.

The function *linkCrypto*(C) works similarly to *link*(C), but it also uses transcoders to enforce encryption. For instance, for an input $C \leftarrow \{C_1, C_2\}$ it produces a $C_1 \rightarrow \textit{transcoder}_{C_1} \rightarrow \textit{transcoder}_{C_2} \rightarrow C_2$ chain. $\textit{Transcoder}_{C_1}$ encrypts C_1's output, and is placed at the same data centre as C_1, while $\textit{transcoder}_{C_2}$ decrypts C_2's input and is allocated at C_2's data centre. The transcoder placement also compatibilises data formats and manage bandwidth utilisation: if C_1 provides a data format x and C_2 expects a different one (e.g., data format y), transcoders are parametrised to realise the $x \rightarrow y$ conversion. One transcoder is chosen to realise the conversion using the same bandwidth-aware logic implemented by *link*(): if x requires more bandwidth than y, the $\textit{transcoder}_{C_1}$ realises the conversion; otherwise, $\textit{transcoder}_{C_2}$.

5.3 Non-encrypted Component Allocation Algorithm (NI)

NI means "non interrupted" and comes from a design premise: when a new endpoint joins the platform, and the content it expects to receive is already being transmitted to other endpoints, the current stream flow cannot be interrupted (e.g., by inserting a new inline tee that distributes the content in a one-to-many fashion). NI implements the non-encryption policy and aims to reduce VM and bandwidth cost by reusing

existent streams as close as possible to the sink endpoint. To minimise the streaming start delay, it reuses already instantiated components instead of instantiating new ones. The NI algorithm is listed in Algorithm 1.

```
1  Algorithm ni()
2    DC_src ← data centre of the source endpoint
3    DC_sink ← data centre of the sink endpoint
4    S ← stream being allocated
5    if finds one tee T at data center DC_sink handling stream S then
6        tee_sink ← T
7    else
8        if finds a tee T at data center DC_src handling stream S then
9            tee_src ← T
10           tee_sink ← new tee at DC_sink
11           link(tee_src, tee_sink)
12       else
13           tee_src ← new tee at DC_src
14           encoder ← new encoder at source endpoint
15           link(encoder, tee_src)
16           tee_sink ← new tee at DC_sink
17           link(tee_src, tee_sink)
18   decoder ← new decoder attached to sink endpoint's compositor
19   link(tee_sink, decoder)
```

Algorithm 1: Pseudocode for the NI algorithm - The algorithm looks for a tee to connect the decoder to. First choice is a tee placed in the same data centre as the sink endpoint (line 5), then one placed in the same data centre as the source endpoint (line 8). If the latter is found, a new tee is allocated near the sink endpoint (line 10). If no tee handling the stream is found, then the whole component chain is allocated (lines 13 to 17). In the end, the encoder is instantiated and connected to the appropriate tee (lines 18 and 19). No encryption is enforced.

Initially, it searches along the stream for tees to connect a new decoder to. The first choice is a compatible tee in the data centre near the sink (line 5), where the decoder is placed. The second choice is a tee in the data centre near the source (line 8), where the encoder is set. If no tee is found, then the content is not being streamed yet, and all required components are provisioned and started (lines 13 to 17). Each involved data centre must contain one tee associated with the allocated stream, a measure that guarantees the absence of interruptions and provides more allocation flexibility.

5.4 NI-PE

The NI-PE algorithm implements the partial encryption policy, and is listed in Algorithm 2. Compared with NI, the difference is the encryption utilised when the transmitted data flows over unsafe zones (e.g., among distinct data centres, interconnected via the Internet). *linkCrypto*() enforces encryption when linking tees placed in distinct data centres (line 11) and when encoders/decoders are located in unsafe zones (lines 21 and 27).

```
1  Algorithm niPe()
2      DC_src ← data centre of the source endpoint
3      DC_sink ← data centre of the sink endpoint
4      S ← stream being allocated
5      if finds one tee T at data center DC_sink handling stream S then
6          tee_sink ← T
7      else
8          if finds a tee T at data center DC_src handling stream S then
9              tee_src ← T
10             tee_sink ← new tee at DC_sink
11             linkCrypto(tee_src, tee_sink)
12         else
13             tee_src ← new tee at DC_src
14             if source endpoint is in a safe zone then
15                 encoder ← new encoder at source endpoint
16                 link(encoder, tee_src)
17             else
18                 encoder ← new encrypted encoder at source endpoint
19                 linkCrypto(encoder, tee_src)
20             tee_sink ← new tee at DC_sink
21             linkCrypto(tee_src, tee_sink)
22     if sink endpoint is in a safe zone then
23         decoder ← new decoder attached to sink endpoint's compositor
24         link(tee_sink, decoder)
25     else
26         decoder ← new encrypted decoder attached to sink endpoint's compositor
27         linkCrypto(tee_sink, decoder)
```

Algorithm 2: Pseudocode for the NI-PE algorithm - Like the NI algorithm, the algorithm looks for a tee close to the sink and source endpoints (lines 5 and 8), and if found near the source endpoint a new tee is allocated and linked closely to the sink endpoint (lines 9 and 10). If no tee handling the stream is found, then the whole component chain is allocated (lines 13 to 21). The main difference is the location-aware encryption policy enforcement when inter-data centre communication is detected, represented by the function *linkCrypto*(). *link*() is still used when the transmission flows through a safe zone, through tests performed on line 14 and 22. Therefore, encryption is partially enforced.

5.5 Fully Encrypted Component Allocation Algorithm (NI-FE)

Also similar to NI, NI-FE is the last proposed component allocation algorithm (listed in Algorithm 3). It implements the full encryption policy, meaning that all data streams are encrypted. Therefore, all encoders and decoders are encrypted (lines 14 and 18), and the usage of *linkCrypto*() when linking components ensures the encryption enforcement for all use cases (lines 11, 15, 17 and 19).

```
1  Algorithm niFe()
2  |  DC_src ← data centre of the source endpoint
3  |  DC_sink ← data centre of the sink endpoint
4  |  S ← stream being allocated
5  |  if finds one tee T at data center DC_sink handling stream S then
6  |  |  tee_sink ← T
7  |  else
8  |  |  if finds a tee T at data center DC_src handling stream S then
9  |  |  |  tee_src ← T
10 |  |  |  tee_sink ← new tee at DC_sink
11 |  |  |  linkCrypto(tee_src, tee_sink)
12 |  |  else
13 |  |  |  tee_src ← new tee at DC_src
14 |  |  |  encoder ← new encrypted encoder at source endpoint
15 |  |  |  linkCrypto(encoder, tee_src)
16 |  |  |  tee_sink ← new tee at DC_sink
17 |  |  |  linkCrypto(tee_src, tee_sink)
18 |  decoder ← new encrypted decoder attached to sink endpoint's compositor
19 |  linkCrypto(tee_sink, decoder)
```

Algorithm 3: Pseudocode for the NI-FE algorithm - Differing from NI-PE, no tests are performed to differentiate safe and unsafe zones, as all traffic is encrypted. Therefore, similar logic to NI is utilised but replacing $link()$ calls with $linkCrypto()$ (lines 11, 15, 17 and 19) to guarantee that all flows are properly protected.

5.6 Container Placement Algorithms

The last proposed heuristics are the container placement algorithms. They are implemented by the orchestration layer presented in Sect. 4 and organise how pods are distributed in the orchestrated nodes to increase VM resource utilisation efficiency and reduce costs. Therefore, the container placement algorithms are complementary algorithms implemented when orchestrating containers.

5.6.1 Best-Fit-Based Algorithm (BFIT)

BFIT, listed in Algorithm 4, implements a well known bin-packing heuristic, best-fit [25], and makes allocation decisions based on CPU utilisation. It receives a list of components to allocate and efficiently distribute them to a set of Kubernetes nodes (lines 10 and 11). If no suitable node is found, a new one is requested and prepared (lines 14 to 17). Once a node is chosen, one of its pods is selected (lines 12 and 18) and inserted in a response list to be returned by the algorithm (lines 13 and 19). The best-fit logic is implemented between lines 6 and 11, which separate utilised and non-utilised nodes, sort them by the level of utilisation, and chooses the one that better fits the allocation candidate component.

```
1  Algorithm bfit(input)
2      input ← list of components to allocate
3      pod_count ← parametrisable amount of pods to pre-allocate
4      response ← empty pod set
5      for each component c ∈ input do
6          used ← subset of kubernetes nodes currently in use
7          empty ← subset of non-utilised kubernetes nodes
8          sort N by unallocated CPU
9          pod ← null
10         for each node n ∈ {used ∪ empty} do
11             if c fits in n then
12                 pod ← reserve pod from n
13                 response = response ∪ {(c, pod)}
14         if pod = null then
15             request and wait for a new VM v
16             add v as a new Kubernetes node n
17             pre-allocate pod_count pods in n
18             pod ← reserve pod from n
19             response = response ∪ {(c, pod)}
20     return response
```

Algorithm 4: Pseudocode for the BFIT algorithm - The workload, a list of components, is received as input. Then, the available Kubernetes nodes are split between in use or empty (hosting no applications), and sorted according to the level of utilisation, implementing the traditional best-fit logic (lines 6, 7 and 9). In the proper order, the algorithm tries to fit the workload in the sorted list of nodes (lines 10 and 11). If no suitable node is found, a new one is requested and prepared (lines 14 to 17). Once a node is chosen, one of its pods is selected (lines 12 and 18) and inserted in a response list to be returned by the algorithm (lines 13 and 19).

5.6.2 Randomised Algorithm (RAND)

RAND, listed in Algorithm 5, goes over a randomised list of nodes and chooses the first one that fits the component under allocation. For each component, a pod from the selected node is inserted in a list, which is returned by the algorithm after allocating all components. New VMs are requested if no suitable node is found, preparing new Kubernetes nodes and re-trying the allocation until all components are in place. The difference from BFIT resides in the node organisation method (lines 6 and 7): instead of sorting nodes by non-utilised resources, the list is randomised, and the algorithm selects the first that suits the allocation-candidate component. This more naive approach is utilised for comparison purposes.

```
1  Algorithm rand(input)
2      input ← list of components to allocate
3      pod_count ← parametrisable amount of pods to pre-allocate
4      response ← empty pod set
5      for each component c ∈ input do
6          nodes ← kubernetes node list
7          randomise nodes by unallocated CPU
8          pod ← null
9          for each node n ∈ nodes do
10             if c fits in n then
11                 pod ← reserve pod from n
12                 response = response ∪ {(c, pod)}
13         if pod = null then
14             request and wait for a new VM v
15             add v as a new kubernetes node n
16             pre-allocate pod_count pods in n
17             pod ← reserve pod from n
18             response = response ∪ {(c, pod)}
19     return response
```

Algorithm 5: Pseudocode for the RAND algorithm - Similarly to BFIT, the workload, a list of components, is received as input. The main difference is that the best-fit logic is not implemented: instead of sorting nodes by utilisation, they are randomly organised (line 7). After this, the logic is the same: the algorithm tries to fit the workload in the randomised list of nodes (lines 9 and 10). If no suitable node is found, a new one is requested and prepared (lines 13 to 16), and once a node is chosen, one of its pods is selected and inserted in a response list to be returned by the algorithm (lines 12 and 18).

6 Components Prototype Evaluation

In this article, we propose software solutions for an A/V collaboration platform, which hosts software components in dedicated VMs or in orchestrated Kubernetes pods. This section describes A/V component prototyping results that fed our simulation framework.

6.1 Virtual Wall Testbed

The Virtual Wall [36] testbed was used to evaluate the prototyped components. The testbed is composed of hundreds of nodes interconnected via gigabit ethernet switches. VMs and dedicated servers can be dynamically deployed and automatically installed, and multiple network topologies can be created among the nodes. The Virtual Wall integrates with the North American GENI [37] and the European Fed4Fire+ [4] initiatives to offer larger distributed topologies.

6.2 Video Properties

As detailed in Sect. 3.3, platform components process A/V data flows encoded using distinct standards (e.g., codec, resolution), providing functionalities such as transcoding, encoding, decoding, one-to-many stream distribution and video-in-video stream composition. For evaluation purposes, we limited the supported resolutions to 256 × 144, 640 × 360, 1280 × 720, and 1920 × 1080. Non encrypted streams are encoded/decoded using the H.264 AVC codec, or kept unmodified (raw), and transported by RTP. Encrypted streams use the DTLS-SRTP standard (AES with 128-bit keys and 80-bit HMAC-SHA-1 authentication) instead of RTP.

6.3 Prototyped Components

Encoders are parameterised in terms of resolution and frames per second to read raw A/V files and encode/encapsulate/encrypt the data and send it to the next component in the stream workflow. The decoder performs the opposite operation, removing encapsulation and encryption, decoding data and providing displayable raw A/V data to a sink, e.g., a screen, a compositor.

Transcoders receive and provide any combination of non-raw video formats (encrypted or not-encrypted), and have a data scaler at the component's core (e.g., it can receive 1920 × 1080 RTP and provide a 640 × 480 DTLS-SRTP transformed output). The last of the prototyped components, tees are designed to handle non-encrypted streams. They receive one input stream and replicate it to multiple outputs that keeps codec and resolution unchanged. Tees do not implement encryption functions to reduce the possible variations of this component (as multiple inputs and outputs would generate an extensive set of customised versions). Transcoders are attached to their inputs/outputs to perform format conversion and encryption operations and counter this limitation.

In preparation for the proposed Kubernetes-based orchestration architecture, components must be containerised and modelled as Kubernetes pods. Utilising Docker, the standard containerisation technology utilised by Kubernetes, components were ported to a minimalist container image, and this image was utilised to compose Kubernetes replicable pods. A new component feature was included to make pod's pre-deployment feasible: when loaded, new pod replicas automatically run the component container and the component binary, which remain standby waiting for additional parameters. These parameters are sent when the pod is selected to be used. Component source code and the data set obtained during the experiments described in this section are publicly available [38].

6.4 Evaluation Methodology

The evaluation took place using a set of VMs hosted by the Virtual Wall testbed: three VMs prepared to host components, and an extra fourth one prepared to send orchestration commands and collect statistics. Each VM allocated to these experiments consisted of an Intel Xeon CPU with 2.4 GHz (6 cores) running Ubuntu 16.06 with 16 GB of

RAM and at least 50 GB of storage. When evaluating the solution based on dedicated VMs to host components, no containerisation or container orchestration was utilised. For the evaluation of unorchestrated containers, a Docker container system was installed on the VMs. Finally, the orchestration of containers analysis introduced a Kubernetes platform: three VMs were configured as Kubernetes worker nodes and the fourth as the Kubernetes master node (which controls the cluster and the container orchestration).

As workload, the video sample Big Buck Bunny [39] was employed to evaluate the prototyped software components, an open movie project licensed under the Creative Commons Attribution 3.0 license [40]. To cover the necessity of several distinct data formats flowing through the software components, the sample was converted to multiple distinct resolutions in raw format: 144 P, 360 P, 720 P and 1080 p. This conversion was done using the avconv solution (Libav library [41]). Components' resource usage were measured during the evaluation rounds by utilising Python scripts [42] that orchestrate component execution, checkpoints inserted into components' code to measure processing and component instantiation delay, and the top command for CPU and memory usage (a Linux command-line tool part of procps, the/proc file system utilities [43]). Memory usage was focused on components' processes for Dedicated VMs and Docker processes when using containers.

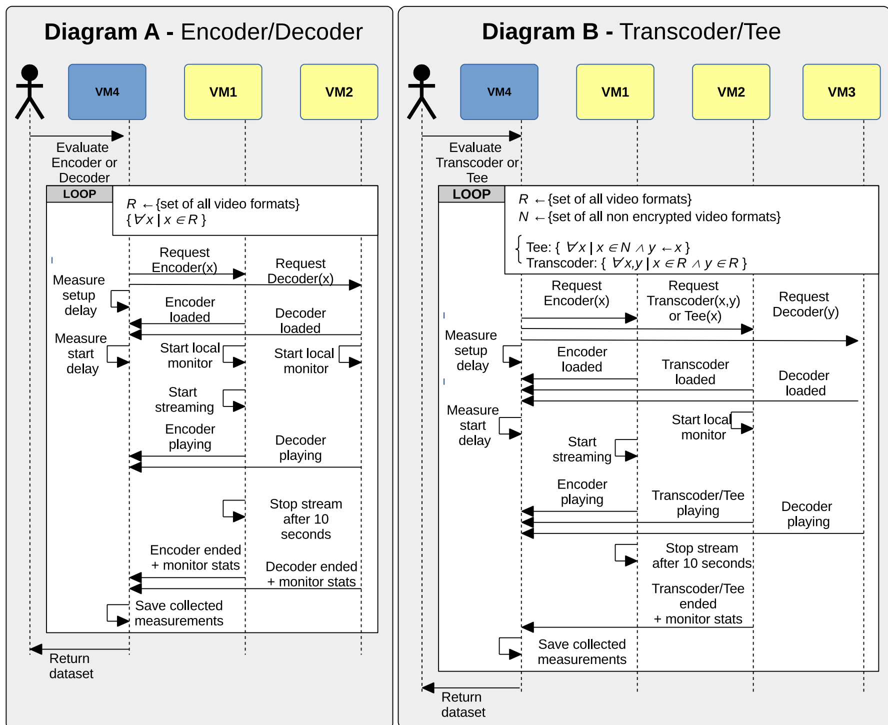

Fig. 7 Message diagrams detailing the evaluation steps for each kind of component: diagram A details the methodology for encoder and decoders, while diagram B does the same for transcoders and tees

The diagrams shown in Fig. 7 depict how the components were instantiated and measured. Diagram A depicts the evaluation methodology for encoders and decoders. For all available video formats, encoder and decoder requests are sent and the instantiation time measurement is immediately started. When the playing status is returned, local monitors start statistics measurements. After 10 s of playback time, components and monitors are stopped and the results data set is compiled. Diagram B shows a similar methodology, utilised to measure transcoders and tees. Transcoder evaluation is repeated for all video format combinations, since they convert stream data from one data format to another, while tees evaluation is repeated for all non-encrypted video formats (as tees process only non-encrypted A/V data). In diagram B, encoders and decoders are used to support the evaluation, but only tees and transcoders statistics are collected. Python [42] controlled all evaluation steps for a total of 50 evaluation rounds, resulting in the data set presented next.

One concern of the solution based on orchestrated containers is the overhead of the Kubernetes platform on the VMs, as Kubernetes components run on each node and pods are kept standby to be chosen and allocated. Our test has shown that this usage is minimal and does not impact the evaluation efforts proposed in this article (comprising less than 1% of the host's total CPU utilisation, a value measured while experimenting distinct workloads combinations). The platform also requires Kubernetes' management components running on separate hosts (the additional master node). We make an assumption that this will be required even if not using Kubernetes. As such, we do not include this back-end cost as an evaluation result and focus exclusively on the resources used by the hosted components.

6.5 Component Performance Evaluation Dataset

The presented results comprise components resource usage statistics, such as memory, CPU, bandwidth, and processing delay. Table 2 presents the first data entry, the bandwidth measured in the encoder output for each of the video formats that can be used between the components. The results are presented by scenario (dedicated VMs, unorchestrated containers, and orchestrated containers), and for encrypted and non-encrypted components.

Table 2 Bandwidth average usage per A/V setup

A/V setup	Encryption	Average (Mbit/s)
256 × 144 H264 AVC	None	0.6713
256 × 144 H264 AVC	DTLS-SRTP	1.2798
640 × 360 H264 AVC	None	2.8268
640 × 360 H264 AVC	DTLS-SRTP	3.3621
1280 × 720 H264 AVC	None	9.9559
1280 × 720 H264 AVC	DTLS-SRTP	10.4003
1920 × 1080 H264 AVC	None	18.2087
1920 × 1080 H264 AVC	DTLS-SRTP	18.3463

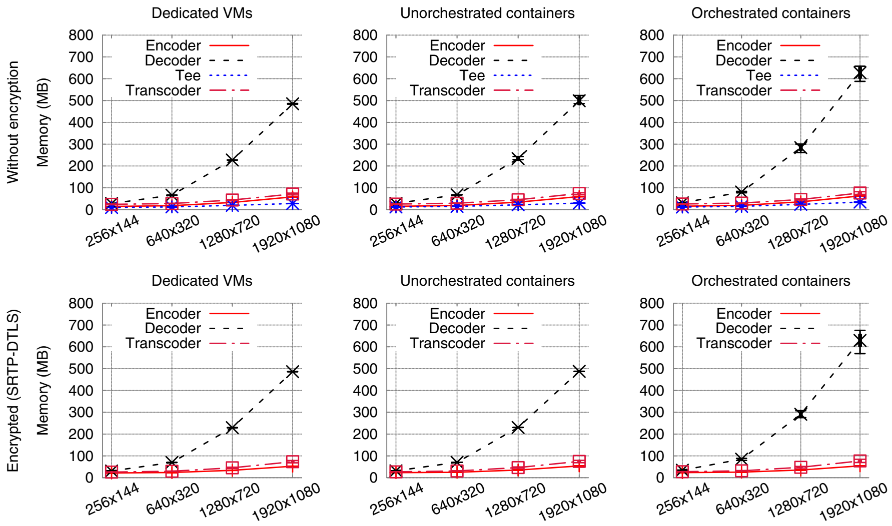

Fig. 8 Encoder, decoder and transcoder memory usage (MB)

Memory usage statistics are depicted in Fig. 8. While most components present low memory consumption, the decoders have high memory needs. To make the automated experiments feasible, we utilised encoder with a dummy sink feature (i.e., a sink device that processes the received raw data but discards it), and we relate this high memory usage to the sink management done by GStreamer [44]-based decoders, the library utilised to build the A/V components.

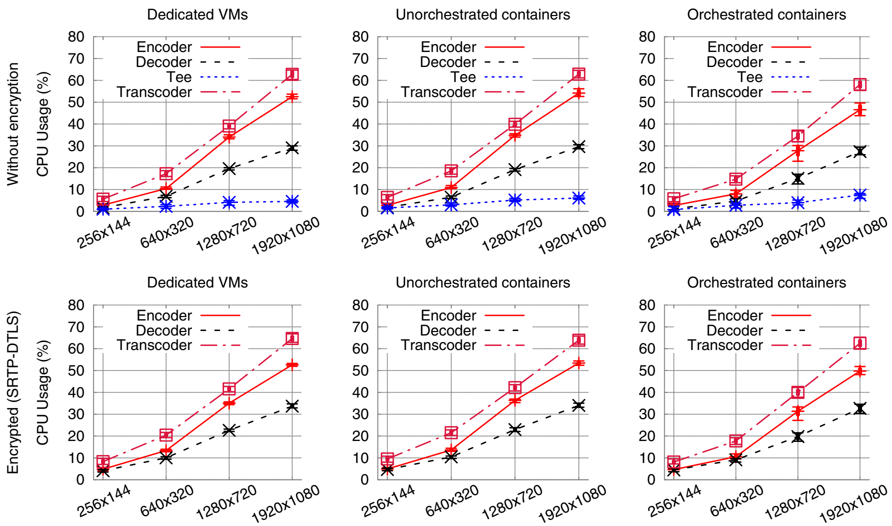

Fig. 9 Single core CPU usage (%) for encrypted and unencrypted A/V streaming

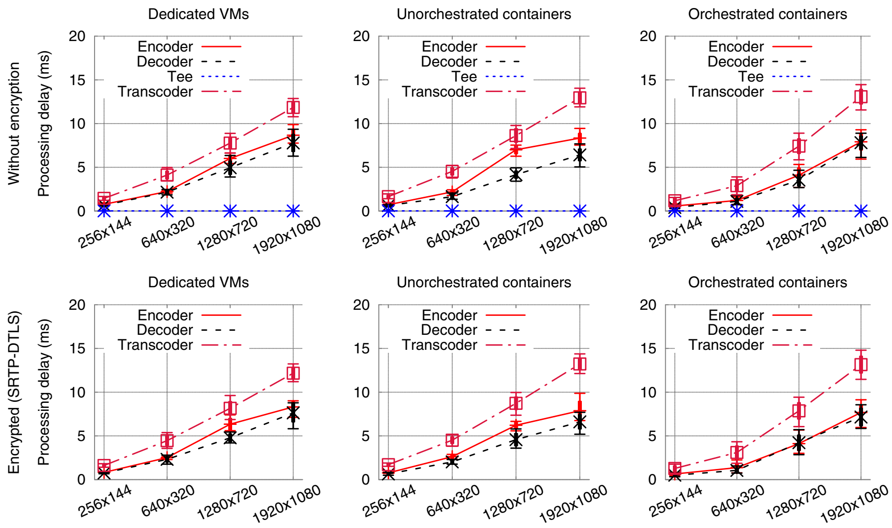

Fig. 10 Processing delay (ms) imposed by components for encrypted and unencrypted A/V streaming

The measured CPU usage is depicted in Fig. 9. The presented values represent the usage of one CPU core (the VMs utilised in these experiments have 6 cores, providing 600% CPU power in total). The results show no CPU overhead due to containerisation or orchestration.

The next result shown in Fig. 10, is the processing time (time taken by an A/V frame from the component input to its output). Values stay stable among dedicated

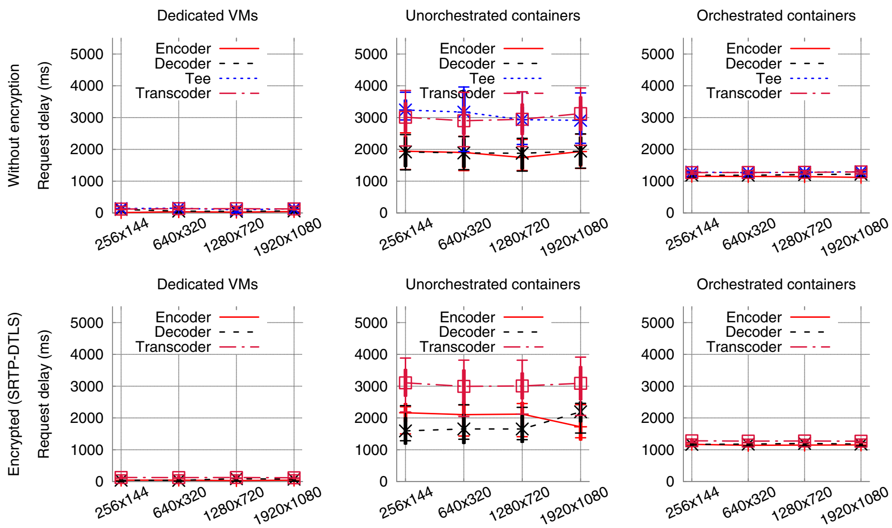

Fig. 11 Time to request components for encrypted and unencrypted A/V streaming: time spent from component request to the moment it is loaded and ready to be used (but not processing A/V data yet)

VMs, unorchestrated containers and orchestrated containers, with no containerisation or orchestration overhead.

The first result that shows significant containerisation overhead is the component request time, shown in Fig. 11. It represents the time spent from the component request to the moment it is loaded and ready to be used (but not processing A/V data yet). As expected, dedicated VMs present low values, as it utilises no containerisation or orchestration. Unorchestrated containers require a Docker container for each instantiated component, a containerisation overhead visible in Fig. 11. Finally, orchestrated containers present no significant containerisation overhead in terms of request delay due to the pre-allocation of pods. However, an orchestration overhead shows up: the request handling performed by the Kubernetes platform takes around 1 s to reserve a pre-allocated pod.

The next data entry is the component initialisation delay: time taken from the moment the component is ready to be used to when it starts to stream A/V content (i.e., acquires a "playing" status). This is shown in Fig. 12. Orchestrated containers present slightly higher values than dedicated VMs, while the utilisation of unorchestrated containers shows higher measurements then dedicated VMs. We relate these higher values to the fact that unorchestrated containers load containers on demand, instead of pre-loading them as done by orchestrated containers (which load containers beforehand when pre-allocating pods).

As detailed in Sect. 3, new stream sessions should be established in up to 2 s. Therefore, component request and initialisation times must be kept in check (e.g., if a component takes longer than 2 s to be load, it will not comply with the deadline and will jeopardise the user experience). Unorchestrated containers presented non-acceptable delays in Figs. 10 and 12, and this drove our design decision of proposing

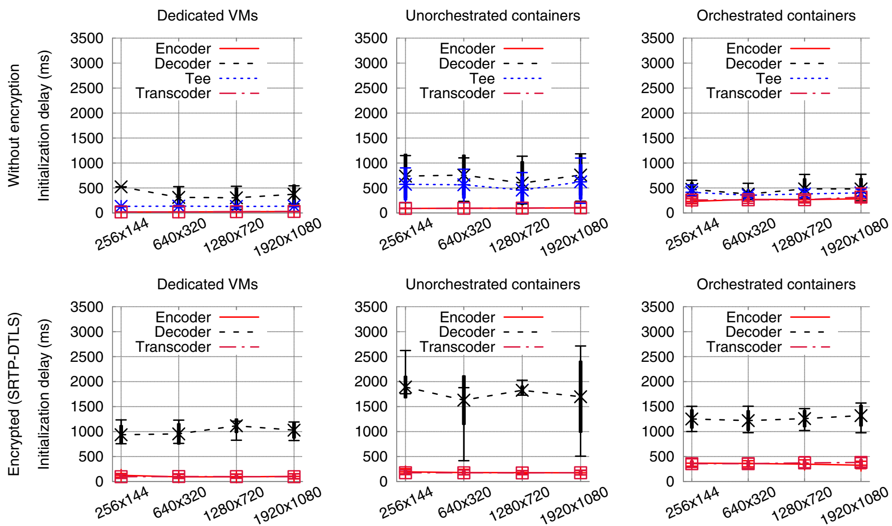

Fig. 12 Time taken from the moment the component is ready to be used to when it starts to stream A/V content

only two software solutions (previously presented in Sect. 4): one based on dedicated VMs and the other on container orchestration. These solutions along with the proposed resource allocation heuristics are evaluated in the next section, utilising the performance evaluation data set presented in this section as input for large-scale collaboration simulations.

7 Discussion of Large-Scale Simulation Evaluations

We investigate a scenario with multiple instructors and instruction attendees streaming training material and attendees' feedback (i.e., two-way collaboration). The presented results show the efficiency of the proposed heuristics and software solutions in this scenario.

7.1 Evaluated Algorithm Combinations

One type of VM allocation algorithm is utilised for all simulations described in this section: RATE+HISTORY. The component allocation and container allocation algorithms listed in Sect. 5 complement each other when orchestrating containers i.e., besides deciding the allocation of the components, the solution that orchestrates containers needs the container allocation algorithms to organise pods among all worker nodes efficiently. However, the proposed solution based on dedicated VMs does not use containerisation and does not implement container allocation algorithms. Therefore, component allocation algorithms and container allocation algorithms are combined depending on the features to be simulated. For instance, when orchestrating containers, NI-PE-BFIT represents the combination of the NI-PE component allocation algorithm with the BFIT container allocation algorithm. On the other hand, NI-FE-NS, utilised when dedicating VMs, utilises the component allocation algorithm NI-FE, but with no container allocation algorithm combined. All simulated algorithm combinations are listed in Table 3.

Table 3 Algorithm combinations

Software solution	Algorithm combination	Component allocation	Container allocation
Based on orchestrated containers	NI-FE-BFIT	NI-FE	BFIT
	NI-FE-RAND	NI-FE	RAND
	NI-HE-BFIT	NI-PE	BFIT
	NI-HE-RAND	NI-PE	RAND
	NI-BFIT	NI	BFIT
	NI-RAND	NI	RAND
Based on dedicated VMs	NI-FE-NS	NI-FE	–
	NI-HE-NS	NI-PE	–
	NI-NS	NI	–

7.2 Simulation Setup

A collaboration scenario generation tool was previously developed taking into account industrial partners' feedback [1]. We extended it to accept encryption parameters, and seeded it with the prototype evaluation results presented in Sect. 6. Component types and streaming formats are listed in Sect. 6.2 and depicted on Fig. 2, respectively. The usage pattern consists of a one training instructor and up to 100 dynamically distributed attendees i.e., local endpoints (located at the instructor's location), remote endpoints (located at remote instruction rooms), mobile endpoints and home endpoints. All endpoints can receive and provide content simultaneously.

Amazon EC2 templates [45] and cost model were used for component hosting and cost calculation. The VM templates and costs configured in the simulator are listed in Table 4. One vCPU indicates one CPU core, with a minimum guaranteed performance based on a CPU credit system [46], called vCPU baseline. Higher bursts are possible but not guaranteed, and we consider this baseline as the VM capacity to meet near real-time demands and avoid quality of experience losses.

Table 4 also shows an estimation of guaranteed cores (e.g., t2.2xlarge has 8 vCPUs and 17% as vCPU baseline, resulting in 1.36 guaranteed vCPUs available). To make the Amazon, Virtual Wall and Cloudsim in line with each other, we consider that one vCPU core provides a fixed total of 1000 Millions Instructions per Second (MIPS), the processing unit used by the CloudSim simulator. Consequently, t2.2xlarge provides 1360 MIPS. For instance, a component that used 50% of one VirtualWall core during the prototype evaluations detailed in Sect. 6, requires 500 MIPS from CloudSim, fitting in the t2.large template.

As explained when proposing software solutions (Sect. 4), the one based on dedicated VMs runs one component per VM and depends on multiple distinct VM templates. Therefore, all templates listed in Table 4 were used when simulating this solution. The second solution, based on orchestrated containers, runs multiple components per VM, which hosts Kubernetes worker nodes, and reduces costs. Consequently, only one VM template is utilised when orchestrating containers: t2.2xlarge.

An Ubuntu 14.04 VM, with 24 GB of RAM and 8 virtual Intel Xeon E5-2630 CPU cores (2.3 GHz), hosted the simulator framework. The metrics to evaluate the performance of the proposed heuristics are detailed next.

Table 4 Amazon EC2 templates used as baseline to simulate and calculate costs

Template	vCPUs	vCPU baseline (%)	RAM (GiB)	Cost per month ($)	Guaranteed vCPUs	MIPS
t2.nano	1	5	0.5	4.91	0.05	50
t2.micro	1	10	1	9.81	0.1	100
t2.small	1	20	2	19.62	0.2	200
t2.medium	2	20	4	39.24	0.4	400
t2.large	2	30	8	78.48	0.6	600
t2.xlarge	4	23	16	156.95	1.0	900
t2.2xlarge	8	17	38	313.89	1.36	1360

7.3 Evaluation Metrics

Total VM cost ($): this metric summarises the costs of all VMs utilised during the simulation. For a simulated scenario composed of D data centres, with V_d the full set of VMs running in a data centre d, $C_{d,v}$ the cost of a VM v when running in a data centre d, and with S_v and E_v giving the time of boot and shutdown of a VM v, respectively, the result is given by Eq. 1.

$$\sum_{d=1}^{D}\sum_{v=1}^{V_d}(C_{d,v} \times (E_v - S_v)) \tag{1}$$

Total VM cost per user ($): this metric summarises the VM costs per user, and better represents the operational costs of the proposed solutions under varied user demand. For instance, when simulating 50 users, the measured value will be *Total VM cost ($)*/50.

Stream placement delay (s): the time spent to start a stream, from component deployment to the first transmission of data. This delay is composed of two parts: processing delay (the time spent by components to output the data after it is received as input) and setup delay (time that is taken to initialise a component after it is requested). A stream workflow composed of four components (encoder, transcoder, tee, and decoder) is used to exemplify the placement delay in Fig. 13.

Components are loaded individually, with loading time defined as the sum of request time, resource allocation time (heuristics decisions) and component setup time (time to load and configure the component). As components are loaded in parallel (i.e. no cumulative sequential setup times) the loading time considered is the longest among all chained components. On the other hand, the processing time is cumulative: after the setup, components establish interconnections and DTLS handshakes and start data transmissions, which must be taken into account is the longest loading time of all chained components. The information passes through the chained components one by one, with each of them sequentially imposing an internal processing delay. Consequently, the placement delay ($D_{placement}$) is the sum of the processing delay ($D_{processing}$) and the setup delay (D_{setup}). For a chain composed of n interconnected components, from c_1 to c_n, with the network delay between two components a and b defined as $nd_{a,b}$, the component request time of component c defined as rt_c, the resource allocation time defined as ra_c, the

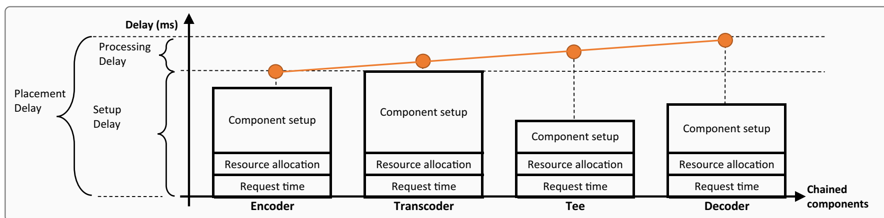

Fig. 13 Example of the delay imposed when provisioning chained streaming components, composed of the delay to set them up and processing delays

component setup time defined as st_c, and the component processing delay defined as pd_c, $D_{placement}$ is given by Eq. 2.

$$\begin{aligned} D_{placement} &= D_{processing} + D_{setup} \\ D_{placement} &= pd_{c_1} + \sum_{i=2}^{n}(pd_{c_i} + nd_{c_{i-1},c_i}) + \max_{i\in(1..n)}(rt_{c_i} + ra_{c_i} + st_{c_i}) \end{aligned} \tag{2}$$

Successful join rate (%): a collaboration meeting join deadline critical to maintaining a good user experience [1]. Therefore, for a set S of streams, a deadline of d seconds, and R_s and S_s representing the request and the start time moments of a stream s, the adherence rate to the deadline is provided by Eq. 3.

$$\frac{1}{|S|}\sum_{s=1}^{S}\begin{cases} 1, & if\,(S_s - R_s) \le d \\ 0, & if\,(S_s - R_s) > d \end{cases} \tag{3}$$

Algorithm runtime (ms): once a stream is requested, the resource placement algorithms must make decisions as fast as possible to avoid negatively impacting the successful join rate. This metric indicates the performance of the algorithm with regards to rapid decision making. Algorithms were profiled to extract runtime performance.

VM pool availability: as described when proposing architectures, a pool of VMs is preemptively populated to allocate component chains rapidly. This metric evaluates the amount of VMs kept in the pool.

7.4 Simulation Results

The present results are based on the algorithm combinations listed in Table 3. Figure 14 depicts the simulated behaviour, and the consumed bandwidth: (a) represents the rate of streams generated by endpoints joining and leaving the collaboration

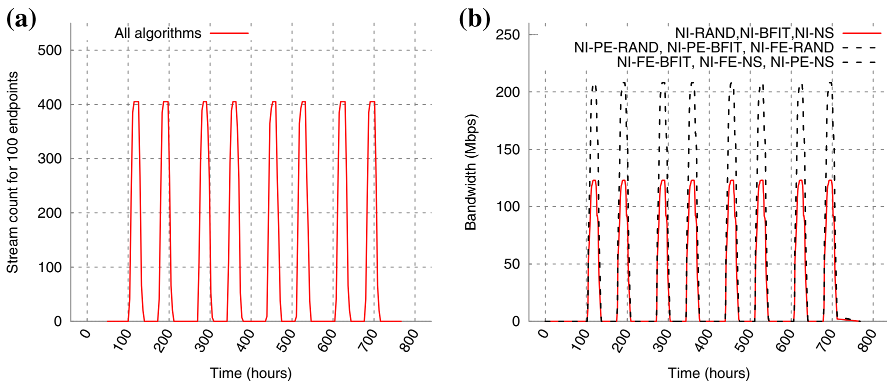

Fig. 14 **a** Collaboration pattern comprising streams for 100 simultaneous users, including cameras from tutor, attendees, training room and presentation material; **b** Bandwidth consumption at instructor's network gateway for the same 100 users collaboration pattern

sessions; and (b) the network traffic measured in the upstream of the instructor, point that concentrates the higher rates in the simulated topology. The session behaves as expected, with two training sessions per week for four weeks. The measured traffic differentiates higher rates for encrypted from lower rates for non-encrypted streams (encrypted formats utilise more bandwidth, as mentioned in Table 2).

The encryption performed by transcoders protects the data, but also requires more resources. Figure 15 shows the number of components instantiated in support of the placed collaboration streams. Transcoder component count is shown in Fig. 15a. The number of transcoders varied based on encryption policies, as explained in Sect. 5. As the enforcement of encryption does not affect the deployment of the other components (encoders, decoders, tees and compositors), they are utilised in the same quantity for all algorithms, as shown in Fig. 15b.

The time to join a collaboration session is depicted in Fig. 16a. This result is related to the component's delays mentioned in Sect. 6.5. The algorithms based on dedicated VMs (NI-NS, NI-FE-NS and NI-PE-NS) do not present container orchestration overheads, resulting in lower placement delays, and consequently lower times to join a

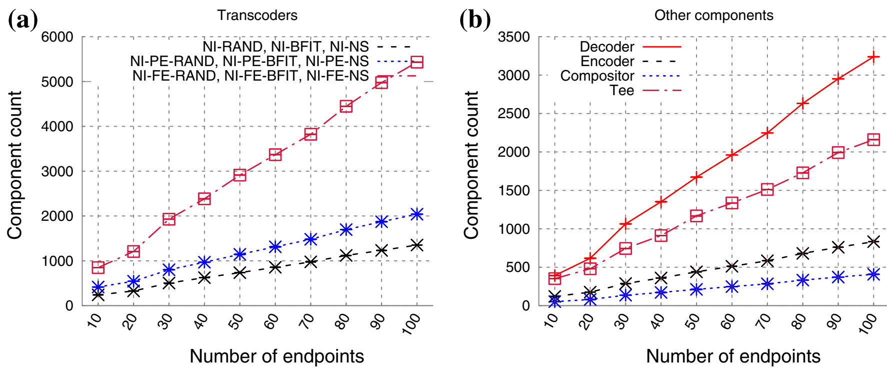

Fig. 15 **a** The amount of transcoders used during the simulation for 100 simultaneous users, which vary according to the enforcement of encryption; **b** The amount of each of the other components

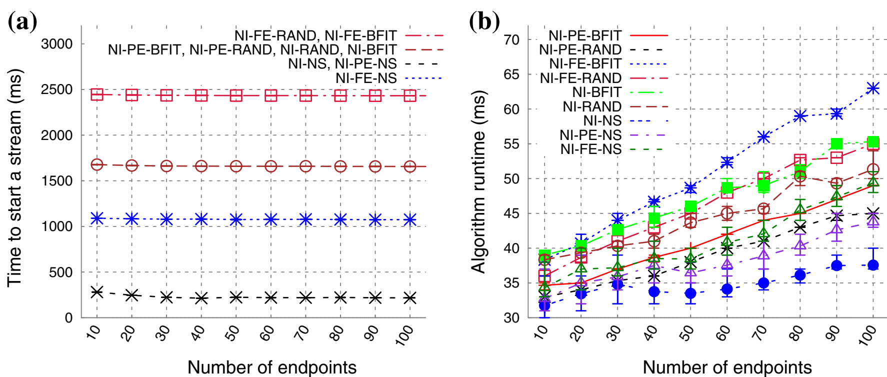

Fig. 16 **a** Time to establish a new streaming session; **b** runtime of the allocation algorithms (ms)

collaboration session. NI-NS and NI-PE result in even lower values, as no encrypted decoders are utilised. NI-FE-RAND, NI-FE-BFIT, NI-FE-RAND, NI-PE-BFIT, NI-PE-RAND, NI-RAND, and NI-BFIT are utilised when orchestrating containers, with an overhead that increases the measured placement delays, and that consequently raises the time to join a collaboration session. NI-FE-RAND and NI-FE-BFIT present values slightly over the 2 s deadline enforced by the EMD research project, pointing to a relaxation of this parametrisable requirement as a trade-off towards more flexibility and lower costs. An important component of the placement delay, and consequently of the time to join a collaboration session, is the algorithm runtime (shown in Fig. 16b). The values grow linearly under acceptable performance for 100 users.

Figure 17 depicts the CPU allocation efficiency of the platform (percentage of utilised/non-utilised VM CPU). RAND algorithms allocate components poorly into VMs, requiring more VMs and is, therefore, less efficient. The other algorithms present efficiency between 50% and 80%, with better results for the BFIT in comparison with the solutions that dedicate VMs. Although with similar VM utilisation efficiency levels, the approach that utilises dedicated VMs requires more VMs (i.e., due to the utilisation of more VM template variations). The quantities of VMs kept in the VM pools are shown next.

Costs are proportional to VM templates' sizes and to the amount of instantiated VMs, as larger templates are proportionally cheaper than smaller ones. The VM management algorithms preemptively populate pools of VMs to host components or Kubernetes worker nodes, and these pools are shown in Fig. 18. As expected, NI-NS, NI-PE-NS and NI-FE-NS, algorithm combinations focused on dedicated VMs, require a larger set of VM templates, and consequently more pre-loaded VMs. The algorithm variations based on the orchestration of containers require less VM templates, and therefore less available VMs. Among them, BFIT variations perform better and require the lower volume of pre-initialised VMs.

Figures 19 and 20 depicts the total costs of VMs and the cost of VMs per platform user, metrics listed in Sect. 7.3 (considering the whole simulation period, one

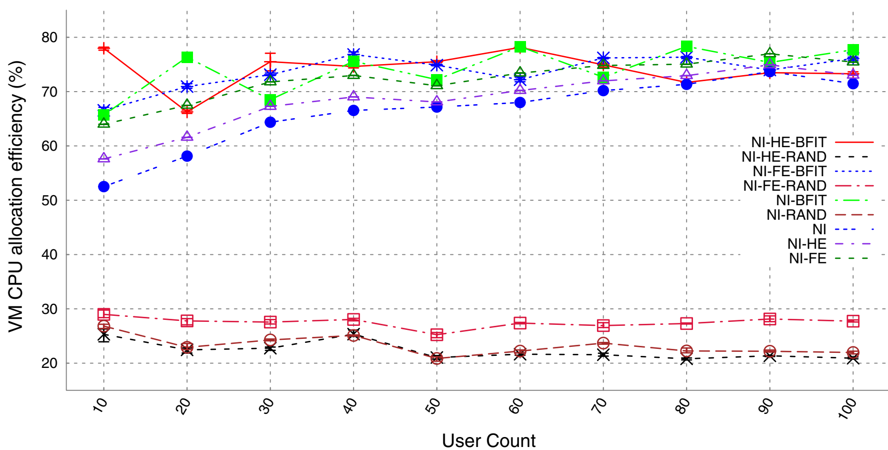

Fig. 17 CPU allocation efficiency for all VMs utilised during the simulation

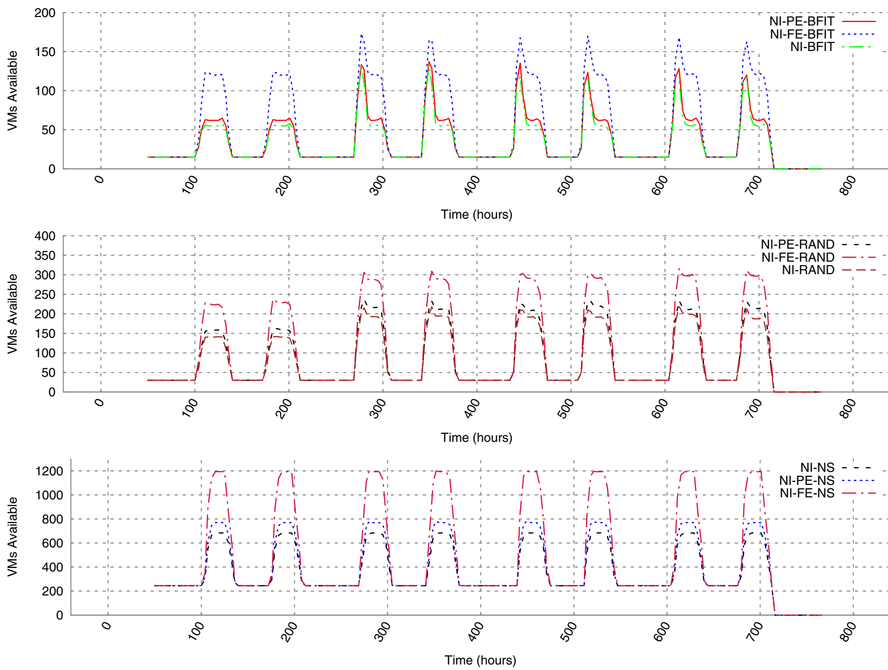

Fig. 18 VMs kept initialised by the VM management algorithms during the whole simulation

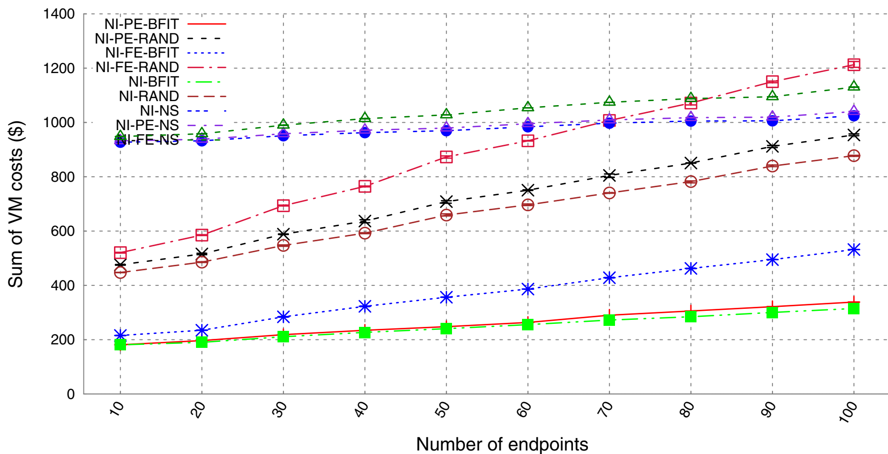

Fig. 19 Total VM cost measure for the whole simulation

month of collaboration sessions). For 10 concurrent users, and among the algorithm variations that orchestrates containers, BFIT-based variations keep less VMs in the VM pools and lowers operational costs ($200, $20/user). The lower efficiency when distributing components among Kubernetes nodes by utilising RAND variations

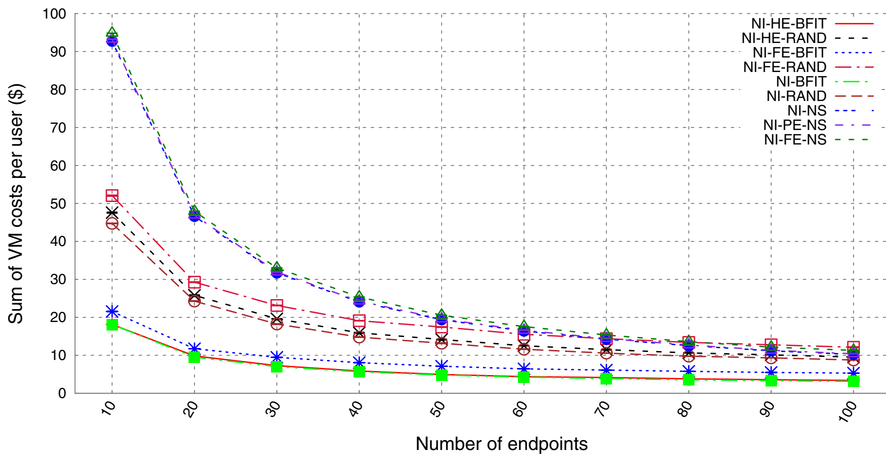

Fig. 20 Per-user VM cost measured for one month of collaboration simulation. The value represents the total VM cost divided by the number of simulated users

require more VMs and increases the average costs and the operational costs for the same 10 concurrent users ($550, $55/user). For the same number of users, the utilisation of dedicated VMs requires a larger set of VM templates, which imposes an operational cost increase ($950, $95/user). When simulating 100 users, BFIT still results in lower costs, evolving non-linearly.

It is also evident in the graphs the connection between the level of encryption, the number of transcoders and the costs. For instance, the BFIT-based algorithm combination NI-FE-BFIT utilises more transcoders to enforce encryption, resulting in higher costs than NI-PE-BFIT, which uses fewer transcoders. The same is valid when comparing NI-PE-BFIT and NI-BFIT: the latter does not enforce encryption, utilises fewer transcoders, and therefore imposes lower costs. For RAND and NS-based variations, the conclusion is the same: due to higher utilisation of transcoders when enforcing encryption, costs increase in this order: NI-FE and NI-FE variations, that encrypt all transmissions, NI-PE variations, that encrypt part of the streams, and NI variations, which encrypt no streams.

8 Conclusion and Future Work

This article proposes cloud provisioning algorithms and software solutions tuned for encryption-enabled professional collaborative media applications. A large-scale collaboration scenario was modelled and evaluated utilising a CloudSim-based simulation framework extended to support the proposed software solutions. As no resource usage statistics were available for the streaming components, these were prototyped and evaluated in support of solutions investigations and larger-scale evaluation by simulation.

The prototyping evaluation investigated three distinct scenarios, one utilising dedicated VMs to host A/V streaming components (without containerisation),

another based on unorchestrated containers, and the last one that orchestrates containers utilising a Kubernetes-based solution. The results showed high component initialisation delays for unorchestrated containers, making this solution unfeasible. Therefore, only dedicated VMs and orchestrated containers were incorporated in the proposed solutions, which were simulated for large-scale collaboration sessions.

The simulation results showed that the algorithms, when utilising the proposed solution that dedicates VMs, support policy-based encryption decisions while keeping project requirements in check. However, w.r.t. cost reduction, the orchestration of containers (which implements best-fit-based complementary algorithms) supports policy-based encryption decisions with lower costs, but slightly increasing the time to join a session to 2.5 s.

Although the workload and prototyping evaluation provided a useful and realistic set of component resource usage data, these results can be improved. Stressing the hosts with more components and different workloads and levels of demand can extend the results to more accurate statistics. Additionally, this paper takes into account only one QoE parameter, the time taken for an endpoint that wishes to join a collaboration session. More detailed QoE assessments are necessary to ensure the user experience quality beyond the joining time. These are not in the scope of this article but are potential future work avenues. Bin-packing algorithm's efficiency is also a challenge to be tackled in future research: we compare BFIT with RAND to obtain heuristics that achieve project goals, but the implementation of more sophisticated approaches can optimise resource usage and cloud resource costs.

Reliability is an essential additional project requirement and algorithms that provide reliability to encrypted collaborative streams supported by orchestrated container-based cloud resources a topic to be investigated. Additionally, the evaluation of serverless solutions and cost models can provide a cheaper solution than standby pods. Finally, the platform proposed by the EMD project aimed to be resilient. Therefore, a solution that tolerates data centre or node failures by providing active and alternative streaming paths will be investigated.

Acknowledgements The research described in this article is partially funded by the imec (EMD) ICON research project and the FWO projects G025615N "Optimized source coding for multiple terminals in self-organizing networks" and G059615N "Service-oriented management of a vitalized future internet".

References

1. Xavier, R., Granville, L.Z., Volckaert, B., De Turck, F.: Elastic resource allocation algorithms for collaboration applications. J. Netw. Syst. Manag. **25**(4), 699–734 (2017). https://doi.org/10.1007/s10922-017-9431-2
2. Hightower, K., Burns, B., Beda, J.: Kubernetes: Up and Running: Dive into the Future of Infrastructure. O'Reilly Media Inc, Newton (2017)
3. Nickoloff, J.: Docker in Action. Manning Publications Co, Shelter Island (2016)
4. Vermeulen, B., Van de Meerssche, W., Walcarius, T.: JFED toolkit, fed4fire, federation. In: GENI engineering conference 19 (2014)
5. Szewczyk, P., Macdonald, R.: Broadband router security: history, challenges and future implications. J. Dig. For. Secur. Law **12**(4), 6 (2017)
6. Sermpezis, P., Kotronis, V., Dainotti, A., Dimitropoulos, X.: A survey among network operators on bgp prefix hijacking. ACM SIGCOMM Comput. Commun. Rev. **48**(1), 64–69 (2018)

7. Cheng, N., Wang, X.O., Cheng, W., Mohapatra, P., Seneviratne, A. Characterizing privacy leakage of public wifi networks for users on travel. In: INFOCOM, 2013 proceedings IEEE. pp. 2769–2777 IEEE, New York (2013)
8. Dilkash, N., Gupta, A., Jain, A.: Real time video encryption for secure multimedia transfer: a novel approach. Int. J. Eng. Sci. Comput. **8**(4), 17077–17080 (2018)
9. Sombatruang, N., Kadobayashi, Y., Sasse, M.A., Baddeley, M., Miyamoto, D.: The continued risks of unsecured public wi-fi and why users keep using it: Evidence from japan. In: 2018 16th annual conference on Privacy, Security and Trust (PST), pp. 1–11. IEEE, New York (2018)
10. Abolghasemi, M.S., Sefidab, M.M., Atani, R.E.: Using location based encryption to improve the security of data access in cloud computing. In: 2013 international conference on advances in computing, communications and informatics (ICACCI), pp. 261–265. IEEE, New York (2013)
11. Bhatti, R., Damiani, M.L., Bettis, D.W., Bertino, E.: Policy mapper: administering location-based access-control policies. IEEE Intern. Comput. **12**(2), 38–45 (2008)
12. Karimi, R., Kalantari, M.: Enhancing security and confidentiality on mobile devices by location-based data encryption. In: 2011 17th IEEE international conference on networks, pp. 241–245. IEEE, New York (2011)
13. Bergkvist A, Burnett DC, Jennings C, Narayanan A, Aboba B (2012) Webrtc 1.0: real-time communication between browsers. Working draft, W3C 91
14. Sivakorn, S., Keromytis, A.D., Polakis, J.: That's the way the cookie crumbles: evaluating https enforcing mechanisms. In: Proceedings of the 2016 ACM on workshop on privacy in the electronic society, pp 71–81. ACM, New York (2016)
15. Al Fardan, N.J., Paterson, K.G.: Lucky thirteen: breaking the tls and dtls record protocols. In: 2013 IEEE symposium on security and privacy, pp 526–540. IEEE, New York (2013)
16. Jennings, B., Stadler, R.: Resource management in clouds: survey and research challenges. J. Netw. Syst. Manag. **23**(3), 567–619 (2015)
17. Koslovski, G., Soudan, S., Goncalves, P., Vicat-Blanc, P.: Locating virtual infrastructures: users and InP perspectives. In: 2011 IFIP/IEEE international symposium on integrated network management (IM), pp. 153–160 (2011)
18. Alicherry, M., Lakshman, T.: Network aware resource allocation in distributed clouds. In: IEEE INFOCOM, pp. 963–971 (2012)
19. Steiner, M., Gaglianello, B.G., Gurbani, V., Hilt, V., Roome, W., Scharf, M., Voith, T.: Network-aware service placement in a distributed cloud environment. SIGCOMM Comput. Commun. Rev. **42**(4), 73–74 (2012)
20. Zhu, Y., Liang, Y., Zhang, Q., Wang, X., Palacharla, P., Sekiya, M.: Reliable resource allocation for optically interconnected distributed clouds. In: 2014 IEEE international conference on communications (ICC), pp. 3301–3306 (2014)
21. ETSI Industry Group: Network function virtualisation NFV. http://www.etsi.org/technologies-clusters/technologies/nfv (2013). Accessed 14 Jan 2020
22. Clayman, S., Maini, E., Galis, A., Manzalini, A., Mazzocca, N.: The dynamic placement of virtual network functions. In: 2014 IEEE network operations and management symposium (NOMS), pp 1–9. IEEE, New York (2014)
23. Moens, H., De Turck, F.: VNF-P : model for efficient placement of virtualized network functions. In: 10th international conference on network and service management (CNSM 2014), pp. 418–423 (2014)
24. The Kubernetes Authors: Managing compute resources for containers. https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/. Accessed 3 Apr 2019
25. Man Jr, E.C., Garey, M., Johnson, D.: Approximation algorithms for bin packing: a survey. Approximation algorithms for NP-hard problems, pp. 46–93 (1996)
26. Soltanian, A., Naboulsi, D., Salahuddin, M.A., Glitho, R., Elbiaze, H., Wette, C.: Ads: adaptive and dynamic scaling mechanism for multimedia conferencing services in the cloud. In: 2018 15th IEEE annual consumer communications & networking conference (CCNC), pp. 1–6 (2018)
27. Chunlin, L., Chuanli, M., Yi, C., Youlong, L.: Optimal media service selection scheme for mobile users in mobile cloud. Wire. Netw. **25**, 1–14 (2018)
28. Xavier, R., Moens, H., Volckaert, B., De Turck, F.: Design and evaluation of elastic media resource allocation algorithms using CloudSim extensions. In: 2015 11th international conference on network and service management (CNSM), pp. 318–326 (2015)

29. Xavier, R., Moens, H., Volckaert, B., De Turck, F.: Adaptive virtual machine allocation algorithms for cloud-hosted elastic media services. In: 2016 IEEE/IFIP network operations and management symposium (NOMS), pp 564–570. IEEE, New York (2016a)
30. Xavier, R., Moens, H., Volckaert, B., De Turck, F.: Resource allocation algorithms for multicast streaming in elastic cloud-based media collaboration services. In: 2016 IEEE 9th international conference on cloud computing (CLOUD), pp. 947–950. IEEE, New York (2016b)
31. Xavier, R., Moens, H., Slowack, J., Sandra, W., Delputte, S., Volckaert, B., De Turck, F.: Cloud resource allocation algorithms for elastic media collaboration flows. In: 2016 IEEE International Conference on cloud computing technology and science (CloudCom), pp. 440–447. IEEE, New York (2016c)
32. Internet Engineering Task Force (IETF): Options for securing RTP sessions. https://tools.ietf.org/html/rfc7201. Accessed 10 Apr 2018
33. Ng, K.F., Ching, M.Y., Liu, Y., Cai, T., Li, L., Chou, W.: A p2p-mcu approach to multi-party video conference with webrtc. Int. J. Fut. Comput. Commun. **3**(5), 319 (2014)
34. Mao, M., Humphrey, M.: A performance study on the vm startup time in the cloud. In: 2012 IEEE 5th international conference on cloud computing (CLOUD). IEEE, New York. pp. 423–430 (2012)
35. ISO I, Std I: Iso 27005: 2011. Information technology–security techniques–information security risk management ISO (2011)
36. Imec Research Institute. jFed Framework. https://jfed.ilabt.imec.be. Accessed 10 Apr 2018
37. Elliott, C., Falk, A.: An update on the geni project. ACM SIGCOMM Comput. Commun. Rev. **39**(3), 28–34 (2009)
38. Rafael Xavier: Prototyped streaming components. https://github.ugent.be/jxavierd/GstreamerComponentsAndDataSet. Accessed 12 May 2020
39. Big Buck Bunny: Big Buck Bunny Video Sample. https://peach.blender.org/about/. Accessed 14 Jan 2020
40. Creative Commons: Creative commons attribution 3.0. https://creativecommons.org/licenses/by/3.0/. Accessed 14 Jan 2020
41. libav. Libav. https://libav.org/about/. Accessed 14 Jan 2020
42. Van Rossum, G., Drake, F.L.: The python language reference manual. Network Theory Ltd (2011)
43. Strauss, F., Wellnitz, O. Procps monitoring tools (1998)
44. GStreamer: GStreamer: open source multimedia framework. https://gstreamer.freedesktop.org. Accessed 10 Apr 2018
45. Amazon Inc. Amazon elastic compute cloud (EC2) images. http://aws.amazon.com/pt/ec2/instance-types/ (2017). Accessed 14 Jan 2020
46. Amazon Inc: AWS CPU credits and baseline performance. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/t2-credits-baseline-concepts.html. Accessed 10 Apr 2018

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Rafael Xavier is a PhD student at the Department of Information Technology of the Ghent University, imec, Belgium. He received his MSc degree from the Institute of Informatics of the Federal University of Rio Grande do Sul, Brazil. His research interests include network management, software-defined networking, network functions virtualization, security, and optimization of cloud-based applications.

Lisandro Zambenedetti Granville is a professor at the Institute of Informatics of the Federal University of Rio Grande do Sul, Brazil. He is co-chair of the Network Management Research Group (NMRG) of the IRTF and president of the Brazilian Computer Society (SBC). His topics of interest include network management, software-defined networking, and network functions virtualization.

Filip De Turck is a professor at the Department of Information Technology of the Ghent University, imec, Belgium. His research interests include telecommunication network and service management, and design of efficient virtualized network systems. He serves as Chair of the IEEE Technical Committee on

Network Operations and Management (CNOM), and is in the Editorial Board of several journals on network and service management.

Bruno Volckaert is a professor of advanced programming and software engineering in the Department of Information Technology (INTEC) at Ghent University and senior researcher at imec. He obtained his Master of Computer Science degree in 2001, after which he investigated network aware Grid service management in his PhD. His current research deals with reliable and high performance distributed software systems for a.o. City-of-Things, UAVs, intelligent railway applications and autonomous optimization of cloud-based applications.

Affiliations

Rafael Xavier[1] · Lisandro Zambenedetti Granville[2] · Filip De Turck[1] · Bruno Volckaert[1]

Lisandro Zambenedetti Granville
lisandro.granville@inf.ufrgs.br

Filip De Turck
filip.deturck@ugent.be

Bruno Volckaert
bruno.volckaert@ugent.be

[1] Department of Information Technology, Belgium, Ghent University-imec, IDLab, Technologiepark-Zwijnaarde 126, 9052 Ghent, Belgium

[2] Institute of Informatics-Federal University of Rio Grande do Sul, Av. Bento Gonçalves, Porto Alegre 9500, Brazil