Enabling the rescheduling of containerized workloads in an ad hoc cross-organizational collaboration

Laurens Van Hoye[1*], Tim Wauters[1†], Filip De Turck[1†] and Bruno Volckaert[1†]

[1]IDLab, Ghent University - imec, Technologiepark-Zwijnaarde 126, Ghent, 9052, Belgium.

*Corresponding author(s). E-mail(s): laurens.vanhoye@ugent.be;
Contributing authors: tim.wauters@ugent.be;
filip.deturck@ugent.be; bruno.volckaert@ugent.be;
†These authors contributed equally to this work.

Abstract

A group of organizations wishing to collaborate urgently, for example in case of a crisis, need to have a way to quickly deploy applications which enable them to speed up a potentially crisis-resolving decision-making process. A cross-organizational Kubernetes cluster, which is orchestrated by a central operator, allows to initiate these deployments in an ad hoc way. Performance issues may however arise at runtime, for example, a video pipeline belonging to a CCTV camera may produce a too low number of frames per second. The ad hoc cross-organizational collaboration case is especially prone to such issues as the set of candidate nodes and the environment in which they run may not be fully known to the operator. This article therefore motivates and describes the usage of a probe swarm architecture, which allows the operator to quickly generate an overview of the resource capabilities of a set of nodes, by executing code fragments locally. The obtained measurements can then enable the operator to decide on rescheduling operations. Evaluation of an illustrative probe swarm intervention shows that the performance of an example application could improve with factor five, ten or hundred when the pod would be rescheduled. This indicates that the proposed probe swarm architecture

This version of the article has been accepted for publication, after peer review (when applicable) and is subject to Springer Nature's AM terms of use, but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: https://doi.org/10.1007/s10922-022-09699-9

may complement other performance bottleneck detection techniques to improve performance of applications that need to be deployed urgently.

Keywords: Cluster, Kubernetes, operator, probes, scheduler, swarm, urgent

1 Ad hoc pod rescheduling in a cross-organizational cluster

The case studied in this article is that of an ad hoc cross-organizational collaboration, more specifically a set of organizations need to collaborate urgently in the case of a time-critical situation. For example, in case of an explosion on a chemical site, the company, local government, police and firefighters need to share information. Another case is that of an equipment builder connecting to the pipelines of manufacturers to quickly analyze and solve machine interruptions. In all cases, a central operator has the control over a cross-organizational cluster, as shown in Figure 1, which allows the deployment of software components into the different network domains to be orchestrated. The central operator thus has the role of cluster administrator. It is responsible for deploying data pipelines in the cross-organizational cluster and is thus fully aware of the data flows and dependencies that exist. It may or may not be part of a data pipeline itself, i.e. it is either one of the collaborating organizations or a facilitating third party respectively, depending on the use case discussed. One particular use case, which is practically relevant, is that of an emergency control room, in which the operator is an experienced crisis manager, managing a dashboard which is the endpoint of each data pipeline in the cluster. The article is written with this scenario in mind. The cluster itself, being a Kubernetes cluster [1], uses the concept of pods to distinguish groups of containerized workloads. An important observation is that, contrary to a regular Kubernetes cluster, the operator does not have a comprehensive overview of the types of nodes that are part of the cluster in such a cross-organizational setup. This set of heterogeneous nodes is composed of different hardware specifications, different network interconnections and different container runtime configurations. Furthermore, unknown background loads may be present on the worker nodes of the different organizations involved in the collaboration. The usefulness of labels indicating static hardware properties (e.g. a node having a high speed storage system like SSDs instead of slower speed physical hard disk drives) may be gone, as different organizations will likely use different key-value pairs as labels. This uncertainty is especially true when the set of nodes may change over time, due to (nodes of) organizations which join or leave the collaboration. An unknowing operator is more likely to select scheduling decisions that lead to performance issues, something which should be avoided, especially in case of an urgent problem-solving process. A few examples of cases in which a performance degradation may be noticed are:

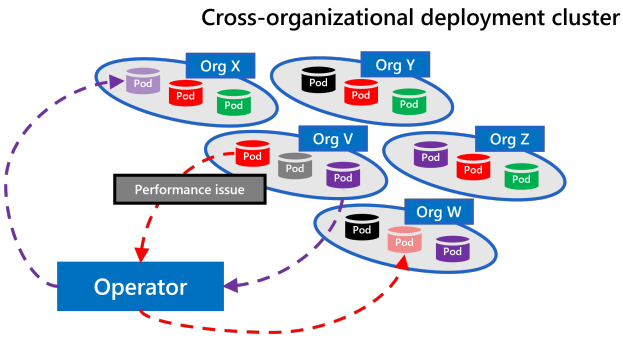


Fig. 1 The operator, having a central overview of all cross-organizational workloads, needs assistance in the rescheduling of misplaced pods.

- A pod running a data-intensive job which suffers from a low-capacity or saturated network link.
- A pod running a storage-intensive job which suffers from slow storage mount options.
- A pod running a job which suffers from the absence of GPU-acceleration.

These kinds of performance issues are all due to scheduling decisions based on limited context information from the environment on which to schedule. The probe swarm architecture presented in this article enables the rollback of these situations by providing insight in the performance capabilities of the different nodes, allowing the operator to decide on urgent pod rescheduling decisions. Note that providing only node-related insight to support a rescheduling review by the operator may prove to be insufficient, as performance issues may have other causes as well, like badly written software, deadlocks, input / output delay, slow human-software interactions, etc. The ultimate goal of the rescheduling of pods is to shorten the execution time required to finish several jobs. Good scheduling decisions may impact the way a critical case is solved. The time dimension is thus of uttermost important in these ad hoc collaboration cases. In this regard, it is also important to note that the goal is not to optimize scheduling decisions, as this would require quite a lot of pod movements, causing unnecessary delays. Providing support for scheduling decisions in a (partially) unknown resource environment, based on time measurements, is the goal of the research presented in this article. The outline is as follows, Section 2 presents work related to the scheduling of workloads and advances for Kubernetes, Section 3 then presents shortcomings in Kubernetes for the scheduling of pods in a cross-organizational setup (first contribution), while Section 4 discusses the composition of a probe swarm to solve the aforementioned problem (second contribution). The evaluation of an illustrative probe swarm intervention is discussed in Section 5 (third contribution), after which the article is finalized with a conclusion and directions for future work in Section 6.

2 Related Work

The contribution presented in this article is part of the FUSE research project [2]. The goal of this project is to enable organizations to collaborate in an ad hoc way by constructing a cross-organizational service mesh. Goethals et al. [3] show which software components are needed to initiate such a cross-domain federation in an ad hoc way. The proposed federation allows cross-organizational deployments to be realized in a few minutes at most. More extensive research is available on the federation of scientific computing environments. Although these types of federations have a less ad hoc character, there are similar challenges to overcome, like the enforcement of diverse local organizational policies. Wickboldt et al. [4] discuss a platform which shortens the time an experienced operator and inexperienced end-users, such as companies, need to provision ad hoc cross-domain network circuits, mainly through the application of easy-to-use visual editors. This scenario closely resembles the case discussed in this article, especially because of the mix of manual and automatic decisions that is inevitably present within ad hoc processes.

Two other topics are already addressed by the authors in previous work. Both consider trust issues that arise when different organizations need to collaborate and thus share data. First, a logging mechanism is needed to allow an honest organization to protect itself against potentially malicious partners and to gain trust in the collaboration at hand [5]. Second, it should be possible for an organization to perform checks and balances with respect to container deployments that are suggested for its domain by a potentially malicious external operator [6]. There was still a need to further research solutions to enable these ad hoc cross-organizational collaborations as they have some important characteristics: they are applied in critical situations and should allow the involved organizations to quickly find a solution for an urgency. The scheduling of workloads in such a context is another topic that exhibits specific properties and is therefore discussed in this article.

A multitude of papers present ideas on how the default Kubernetes scheduler should advance. It is possible to build further on the concept of resource requests, either manually by allowing an operator to classify an application based on its resource usage [7], or automatically by means of extending the Kubernetes Vertical Pod Autoscaling feature [8]. An improved scheduler is often required when the heterogeneity of a cluster, for example in the case of fog computing, plays an important role in the performance of an application. Most papers propose solutions which try to make the scheduler aware of a distinct aspect. The focus could be on the minimization of the overall cost of a Kubernetes deployment in a cloud environment [9]. Similarly, the focus could be on reducing the end-to-end latency between applications while maintaining bandwidth requirements [10] [11], and their placement in geo-distributed environments [12]. Another emerging aspect is that of energy efficiency [13] [14], a strategy applied by a specific set of schedulers among the wider group of topology-aware [15] and hardware-aware schedulers. Examples of the latter are a GPU-aware scheduler making use of historic pod executions to speed

up calculations [16] and an Intel SGX-aware scheduler [17]. Another crucial aspect focuses on the real-time utilization of node resources to schedule workloads [18]. This load-awareness is especially important in multi-tenancy cases [19], as interference effects such as cache misses and CPU context switches may lead to performance degradation. There are also papers which try to combine several of those aspects and propose a weighted multi-criteria decision strategy with the goal to optimize workload placement [20] [21]. The scale at which a scheduling algorithm needs to operate is another distinctive characteristic. A category of papers focuses on scheduling algorithms which are backed by queuing theory fundamentals [22]. Those are crucial in very large dedicated data center setups, but are thus less applicable in this ad hoc case. Finally, there is a paper which proposes an architecture that applies measurement probes at the different worker nodes [23], a similar approach as is presented in this article. Bayer et al. suggest the usage of both resource monitoring probes and application-specific probes, the latter to perform security checks or to monitor energy consumption. This scheduling strategy, focusing on local observations, comes closest to the one presented here. However, as with other cited related work, no other research takes into account the cross-organizational aspect and its potential consequences. The fact that the management of nodes is distributed among different organizations is specific for the Kubernetes clusters deployed in the cross-organizational collaboration case. Furthermore, where other scheduling approaches try to steer decisions based on generically applicable metrics, which are perfect for automated scheduling decisions, it is required here to gather higher-level metrics. These should indicate consequences for the collaboration in a more easy to interpret way for the operator such that a performance issue can be solved quickly. To the best of our knowledge, no other work has been conducted addressing specific concerns related to ad hoc workload scheduling, which are discussed in the next section, introduced by a cross-organizational setup.

3 Necessity of probes in a cross-organizational context

The probing concept presented in this article assists an operator in making rescheduling decisions when the cluster layout is largely unknown. The goal of this assistance is to allow collaboration applications to operate efficiently, thus without severe bottlenecks increasing execution time and downgrading quality of service. Before the probes are detailed and discussed, it is necessary to identify why they should be used in the first place. The remainder of this section therefore illustrates why a vanilla Kubernetes cluster is not sufficiently capable to achieve the proposed goal. The necessity of probes may be explained from different perspectives:

- **Requirements from the perspective of the operator**

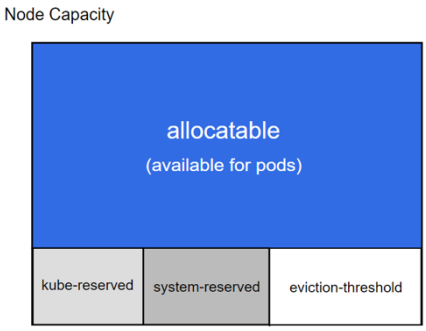


Fig. 2 Overview of node capacity as interpreted by vanilla Kubernetes [26].

- The collaborations considered in this article are of the ad hoc type. The corresponding Kubernetes clusters are thus composed dynamically. This means that the operator is either not or only in a limited way able to reuse any previous knowledge related to cluster layout. For example, a new heterogeneous cross-organizational hardware setup does not have any Kubernetes labels attached that are meaningful for the operator. Furthermore, the operator does not have the time to thoroughly study which resources are offered by the different organizations. Ad hoc cluster management is thus needed for this case, which brings additional difficulties to making proper rescheduling decisions.
- The implementation of the default Kubernetes pod scheduler uses a node filter function, implemented in the `NodeResourcesFit` plugin, which checks pod resource requests against the availability of resources on worker nodes [24]. For example, when a pod requests a CPU time allocation of 1.5 / 4 physical or virtual CPUs, the scheduler will consider 2.5 CPUs to be allocatable for future pods. Figure 2 shows how capacities are represented: except for strictly required daemons, the remaining capacity is considered to be available for pods. The resource availability checks present in the plugin thus only consider the resource requests of pods, not their actual consumption, and most importantly, the load of processes which are not under the supervision of the Kubelet are neglected in the scheduling process [25]. This predictability in container resource consumption and consequently performance, something which is an important aspect in a container orchestrator like Kubernetes, may thus be broken in an ad hoc cross-organizational setup due to the possible presence of severe external background loads and their corresponding unknown effects.
- Only a very limited set of static node properties is available. It is limited to general properties such as the instruction set architecture, kernel version, operating system, Kubernetes version and container runtime version [27]. This set could easily be extended with other properties such as CPU manufacturer, clock speed, hypervisor if the worker node runs as a VM, disk manufacturer, etc. Furthermore, the metrics endpoint of each Kubelet,

`/metrics/resource`, could be used to gather resources metrics of a node over time. A combination of these static and dynamic data could already provide more insight to the operator and solve some ad hoc scheduling questions. There are however two major concerns. The first concern is that it may remain difficult to derive, in an ad hoc way, node performance differences from such an extensive data set. For example, quickly comparing the performance of CPUs from different manufacturers and from different release years, each with their own set of cores, caches, multithreading settings, clock speed values, current workload, physical vs. virtual cores, etc. is almost impossible. The same observation holds for other hardware components such as memory and disk. Furthermore, organizations should be able to dynamically join or leave the collaboration at any point in time, increasing the complexity even further. The second concern is that, due to the fact that the nodes are under management of different organizations, additional node information may not always be available. Organizations may not always approve the sharing of potentially sensitive node information with external entities. Organizations may also decide to only share a restricted view on local node settings. Finally, the sharing of metrics may be limited keeping in mind the associated bandwidth cost. In all cases, it cannot merely be assumed that the operator is able to query every wished-for data.

- ☐ When pods need to communicate according to a cross-organizational flow, e.g. when they constitute a service but are deployed using an anti-affinity rule on an organizational level, it may also be needed to evaluate the performance of the links between the hosting nodes. Severe network bottlenecks may be prevented or discovered this way, allowing the operator to pick another scheduling approach.

■ **Requirements from the perspective of the hosting organization**

- ☐ A hosting organization may impose additional and/or further restricting resource constraints on the containers received from the operator. These local constraints should definitely be taken into consideration for this particular use case, as hosting organizations will highly likely want to protect themselves against potentially malfunctioning or malicious operators claiming most of the available resources. It is important to note that these possible additional resource restrictions are completely unknown to the operator. The theoretically infinite set of static and dynamic data, as suggested above, thus not necessarily forms a sufficient base anymore to allow the operator to schedule properly.
- ☐ These locally enabled resource constraints may be the same for each container that is proposed by the operator, but they may also be different. The Docker container runtime is one example of a runtime which allows resource settings to be configured per container [28]. This dynamic aspect makes it impossible for the operator to know under which conditions the container will run. Examples of such dynamic configurations are: (1)

enabling/disabling swap memory when a container process reaches the memory limit, or (2) configuring that three external containers are allowed to consume half of CPU capacity at maximum, according to a specific partitioning like $\frac{1}{4}$, $\frac{1}{6}$ and $\frac{1}{12}$. It is clear that these examples negatively affect the predictability of container performance, an important reason why Kubernetes did not support swapping to disk until version 1.23, the most recent one at the time of writing [29].

These observations lead to the conclusion that it is necessary for the ad hoc cross-collaboration case to evaluate constraints locally. Probes, which are discussed in the next section, pave the way for an operator to gather more insights supporting a rescheduling decision under these circumstances. Note that probing may not necessarily be required during the entire collaboration duration. When the availability of node resources is identified, and when the cluster operates in a more or less steady and predictable way, most uncertainties are solved.

4 Probe swarms enabling pod rescheduling

This section first discusses probes in general and what they could look like in a few examples. Afterwards, a possible probe swarm architecture with corresponding steps is proposed, providing an idea how probing could be integrated in vanilla Kubernetes.

4.1 Probes as performance indicators

A probe, in its most general definition, is a software function or a collection of functions, thus consuming (a combination of) resources such as CPU, memory, disk, GPU, network bandwidth etc. They are short-lived and finite as there are only a few tens of seconds at maximum between the probe pod starting up and tearing down. It is thus a code fragment which needs to be processed by a selection of worker nodes, allowing an operator to obtain an overview of execution times and thus relative performance differences between nodes. As is clear from this definition, a probe can be selected from an infinite pool of possible functions. Two types of probes can be identified at both ends of the spectrum. The first type of probes, the generic probes, could be applied independently of the workload that needs to be scheduled. These probes allow to dynamically pressurize target resources, as there may be a CPU-intensive probe, a memory-intensive probe, a disk-intensive probe, etc. The output produced by the probe execution is useless for the collaboration, i.e. only the corresponding execution time is important. An example of such a generic probe can be found in the class of algorithms calculating the n-th digit of Pi. One well-known use case of these algorithms is to benchmark compute infrastructure. A relevant implementation is for example the calculator TachusPi [30], which is able to calculate billions of digits using only commodity hardware. The second type of probes are exact copies of the considered workload. The containers of a pod

could simply be duplicated to other nodes in the cluster. Both types of probes have clear disadvantages. The generic probes will likely provide relative performance differences between nodes if the operator can find an appropriate parameter set. However, it may remain difficult for the operator to interpret these results with regard to their impact to the collaboration at hand. A more insightful measurement may thus be handy. The copy probes on the other hand, do resemble the original pods, but their initialization may not be ad hoc and their execution may also severely impact the probed nodes and the processes that are running there. A third option is to consider application-specific probes which are somewhere in the spectrum discussed above. They could respectively allow for a more insightful and more efficient probing solution when the two discussed types of probes are not considered appropriate. Put generally, it would be possible to use derived probes, i.e. representative functions, based on the considered workload. Each workload boils down to an application, being a main function, which could be further decomposed into (much) smaller functions, each which could be used as a probe. Decomposing an application on code level seems impossible for this case, as both reverse engineering a binary and analysing the different functions takes time. It is however possible to select probes, which conceptually match several parts of the workload, from a well-prepared cross-organizational probe catalogue. The granularity of decomposition may differ per case. Theoretically, one could create a service mesh of configurable and linkable probes reflecting a multitude of applications. In reality however, due to time limitations caused by the ad hoc character of the collaboration, a line needs to be drawn between a portfolio of either more general or more specific probes. A balance between reusability and efficiency in time needs to be found.

Two application-specific probes are proposed in this article. Other examples are possible, but these serve to illustrate the idea behind the deployment of reusable probes, that is allowing the evaluation of a certain algorithm which may resemble a pool of possible workloads. This way, different resource consumption patterns can easily be tested on the different nodes.

- **Video processing probe**: The processing of video streams is a frequently reoccurring application in a cross-organizational collaboration. Different camera feeds may be shared in a cross-organizational collaboration, for example to allow the monitoring of an industrial site via static cameras and drones in case of an emergency situation. A screen sharing session is another application which may be used to produce a video stream. For these use cases, different video probes could be defined, e.g. a probe which pre-processes a data source and encodes it according to a video coding standard, and a probe which decodes the stream and does post-processing. These probes could then be parametrized in a such a way that different codecs could be applied, such as H.264/AVC and H.265/HEVC. As these video probes will be CPU-intensive, it might also be possible to evaluate whether a node supports parallelization through multithreading. Even more, it might

be possible to shift some calculations to the GPU and check which performance improvements may be observed from GPU-enabled nodes in the cluster. Note that, contrary to the generic probes, the measured execution time of the probe provides additional insight in resource capabilities. For example, it might be interesting to know how long it takes to encode a video stream of ten seconds, using following command:

```
./encoder.exe --source cam01 --codec h.264 --time 10 --width 640 --
                    height 480 --output encoded.h264
```

- **Data structure probe**: Another frequently reoccurring application is a data storage solution which allows a collection of data to be stored. This data could for example be generated by a video source, a case which would allow for a probing pipeline connecting a video probe with a data structure probe. The collection may be a simple data dump, but mostly a more efficient processing solution is needed. When the collection needs to be stored according to a a specific structure, a data structure needs to be used. Well-studied data structures are for example arrays, linked lists, (binary) search trees and (binary) heaps. They only differ in their implementation of data operations, such as an insertion, deletion, lookup, traversal, sorting, etc. and corresponding asymptotic behaviors. This means that one data structure could easily be swapped for another as long as an interface of functions is implemented. Which structure needs to be chosen depends on which requirements need to be fulfilled, for example the performance of a lookup operation may be more important to that of an addition operation. The performance of these base operations may indicate how suitable a node is to assign and deploy a data storage solution. Again, the absolute execution time of the probe may be of interest here, for example to know how much data could be processed by a single node. Note that both the volatile storage in memory and the persistent storage on disk could be analysed by this specific probe. For the latter case, a tree could for example be written to a file, as is illustrated for a binary search tree (BST) by the C code in Listing 1.

```
// Depth-first traversal to iterate tree with non-empty root
void traversalTree(node* root, funcptr func, meta* meta) {
    if(root->left) { traversalTree(root->left, func, meta); }
    func(root, meta);
    if(root->right) { traversalTree(root->right, func, meta); }
}

// Probe function: write tree to disk
void storeTree(node* root, meta* meta) {
    meta->output = fopen("tree.txt", "w");
    ...
    // displayNode serializes a tree node
    traversalTree(root, displayNode, meta);
    fclose(meta->output);
}
```

Listing 1 A data structure probe should allow to evaluate disk performance of a node as illustrated in this sample.

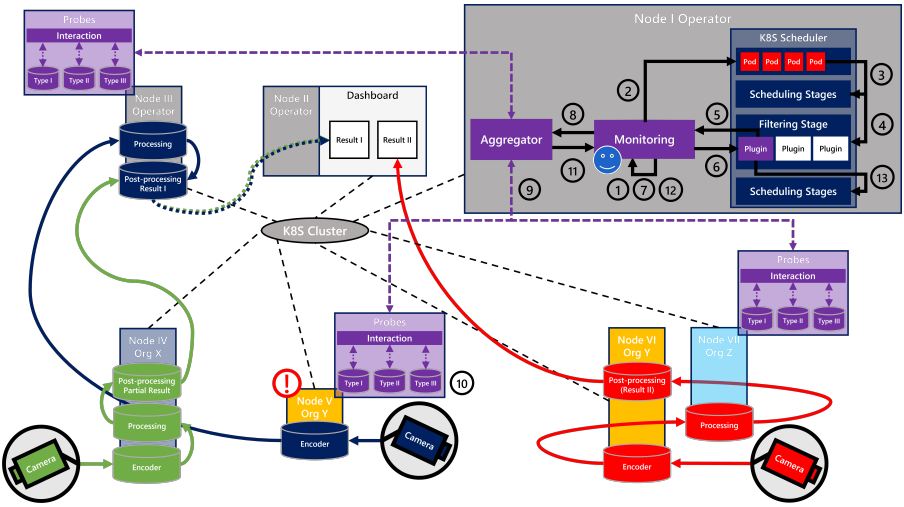


Fig. 3 The architecture needed to deploy the proposed probe swarm.

These probes focus specifically on the resource availability of nodes, but they could also be extended to allow for the evaluation of specific network links. In general, a network probe should allow for communication between two pods hosted at different nodes in the cluster. The integration of such a network probe would allow for more end-to-end based probing tests, for example a video stream could first be pre-processed and encoded at Node X, after which it is sent to Node Y, which then decodes and post-processes the data. This way, it is possible to test whether the throughput of a network link supports the bit rate of a corresponding video stream when it is encoded using a certain parameter set, and whether certain quality of service metrics such as the number of frames per second or signal-to-noise ratio can attain certain wished-for levels.

4.2 Probe swarm architecture

This section will present the flow of a probe swarm architecture, more specifically the steps needed to integrate a probing solution into vanilla Kubernetes. Figure 3 shows the different components. Of particular interest are the purple colored components, because they represent the additional elements that are needed to achieve the proposed goal. The cross-organizational collaboration shown in the figure, illustrates the processing of three camera streams into two relevant results for the case at hand, which are displayed on a dashboard under supervision of the central operator. The steps discussed below assume the case of a pod rescheduling situation caused by performance issues of a node in the cluster.

1. The operator runs a monitoring pod in its own domain, which allows to obtain a complete overview of the cross-organizational collaboration and the corresponding Kubernetes cluster. This boils down to an overview of

all nodes and deployed pods, corresponding resource consumption metrics, data flows between the individual organizations based on a logging solution (e.g. like the one proposed by the authors [5]), and alerts by an alerting system. When an application needed for the cross-organizational collaboration shows issues related to quality of service, an operator may decide to reschedule the corresponding pod. This manual decision may further be supported by automated bottleneck detection techniques, but the exact rescheduling trigger does not immediately matter. In this example, a scheduling issue is present at Node V, being one of the yellow nodes of Organization Y, which will serve as an illustration in the remainder of this section.

2. The operator thus needs to find a solution to move the load of Node V. The Processing pod, responsible for analysing the video stream, may put a too heavy load on this particular node. The operator therefore marks this pod as a candidate to be rescheduled, causing the corresponding YAML file to be sent to the Kubernetes scheduler to be inserted in the pod scheduling queue. Note that there is a dependency between the Processing pod and the Post-processing pod. The operator will first wait for an alternative node to be found for the Processing pod, after which the Post-processing pod can easily be moved to this new destination node. Although this rescheduling may not be necessary from a load perspective, it may be desirable to prevent cross-domain interactions due to accompanying network latencies.
3. Each pending pod is then processed according to a scheduling profile [31]. Such a profile consists of a number of scheduling stages which each have their extension point. Plugins implement either a single or multiple of these extension points. This step represents the pass of the Processing pod through the plugins belonging to the stages before the filtering stage.
4. The filtering stage is constituted of plugins which check for either soft or hard requirements, for example affinity/anti-affinity rules, pod spreading rules, pod resource requests, etc. The suggested probing solution is in fact an additional filtering step, as only nodes with suited performance capabilities should be considered for pod placement. A new scheduling profile should thus be defined consisting of the stages and plugins as used in the default scheduling profile, extended with a custom plugin as the last filtering step. This custom scheduling profile is then available for those pods that require a probe swarm intervention. The Kubernetes Scheduling Framework [32] allows custom plugins, implementing an extension point interface, to be compiled into the scheduler.
5. The custom plugin needs to invoke the monitoring backend of the operator using a webhook. This causes the considered pod and corresponding filtered set of nodes to be registered in the monitoring system and consequently to be presented to the operator. As the plugins in the filtering stage may evaluate nodes concurrently [32], multiple invocations per pod may be expected.
6. The response to the webhook invocation may be either a (1) request registered, response pending notification or (2) a further, by the probing solution,

filtered set of nodes. In the first case, the pod is marked as unschedulable by the custom plugin. This causes the scheduling cycle to be aborted, after which the pod is returned to the scheduling queue waiting for a consecutive cycle to be initiated [32]. Steps 3-6 are thus repeated by the scheduler as long as required. The final attempt is when the second case occurs, i.e. when the scheduling process is able to continue to step 13 with the nodes that passed the probing selection.

7. The operator then investigates the set of proposed nodes. A manual assessment of the filtered set of nodes takes place. There are two possibilities for the pod to be rescheduled:
   (a) The operator tries to reschedule the pod within an organization, so called intra-organization rescheduling. This approach may have advantages. The pod was already allowed to be deployed in the domain, meaning that a switch between nodes in the same domain would not take additional verification time. Furthermore, nodes of the same organization may be most nearby in the network, guaranteeing a more predictable continuation of operation of the pod. Applying this to the discussed example, the operator may first consider Node VI of the same Organization Y. As this node has already offloaded the Processing pod to Node VII of Organization Z, it is clear that this node should be skipped from probe evaluation.
   (b) The operator tries to reschedule across organizations, so called inter-organization rescheduling. This means, again applied to the example, that the operator should select Node III of the operator and Node VII of Organization Z, assuming these nodes were indeed part of the filtered set up to this point, together with the reference Node V, to be evaluated by the probes.

   This manual intervention may thus cause the set of potential nodes to become smaller. It is important to note that node selection is focused on finding an appropriate pod as quickly as possible. The goal is not to search any further for a better scheduling decision, as it would become an optimisation case for the entire cluster, which does not fit the ad hoc and rapidly changing scenario discussed here. This also means that it is not needed to run probes at each node, only at the selected nodes.
8. The types of probes and parameter sets selected by the operator need to be pushed to the aggregator component. A series of commands which need to be executed by the selected nodes are thus communicated in an asynchronous way. Multiple different configurations may be tested over time, enabling the operator to do some live probing.
9. The aggregator is responsible for the deployment of the probes, which are pods themselves. The `kubectl create` command thus needs to be executed. The default scheduling profile is applied to these pods, as probes should be deployed without the intervention of a probe swarm. A probe pod consists of a container which has all binaries required to probe available, and has a process running which keeps the container alive. The aggregator is

then able to push commands, representing the probe executions, to such a container and to obtain time measurements. The `kubectl exec` command could for example be used by the aggregator to enable this. It is of uttermost importance to note that a probe, once selected by the operator via step 8, needs to be executed multiple times with a specified frequency, like every ten seconds. This is needed to filter outliers from the time measurements. This is the reason to keep the probe alive, as otherwise it would be needed to deploy it multiple times, causing an unnecessary overhead for the hosting node. The Processing pod in the example, may represent any video processing step. Which kind of probe is selected by the operator, will thus differ per case. It may range from the deployment of a generic probe to the deployment of a more specific probe executing a computer vision algorithm. The latter is a perfect example of a class of algorithms which could easily be prepared in a probe catalogue. For example, when the processing of the encoded video stream focuses on QR code detection, a probe may easily simulate this as follows:

```
./cv.exe --lib opencv --module objdetect --class qrcodedetector --
                function detect --input encoded.h264
```

10. It is possible to deploy multiple probe instances at a node. These probes may be of the same type, for example multiple video encoders. Such a probing intervention would allow to evaluate how many video streams could concurrently be encoded on a single node. The probes may also be of a different type, for example a video probe and a data structure probe. These different types may even be linked dynamically, for example when it is needed to process video first and to store it afterwards. These interactions allow for more complex probing solutions. Such dynamic links should be prepared as well, to allow the operator to quickly link different probes together.
11. The time measurements are collected by the aggregator, and aggregated for each probe. Aggregation takes place on a rolling basis, i.e. every measurement cycle, the x-th percentile may for example be calculated and passed to the operator. It should be possible for the operator to specify any custom aggregation.
12. Based on both absolute measurements, in case more specific probes are used, and relative differences in probe executions, the operator is able to obtain on overview of the resource capabilities of the different nodes. A weighted evaluation of probe results may be part of this assessment. The operator then manually selects one or multiple nodes, which constitute the new filtered set of nodes. This set of nodes is then passed as a response to the webhook discussed in step 5-6, to allow the scheduling process to progress.
13. The nodes are then further processed by the remaining scheduling stages. Ranking the nodes based on different scoring criteria is the main goal of this final evaluation. For example, nodes which already have the required container image, may be favored. Finally, the most favored node is chosen

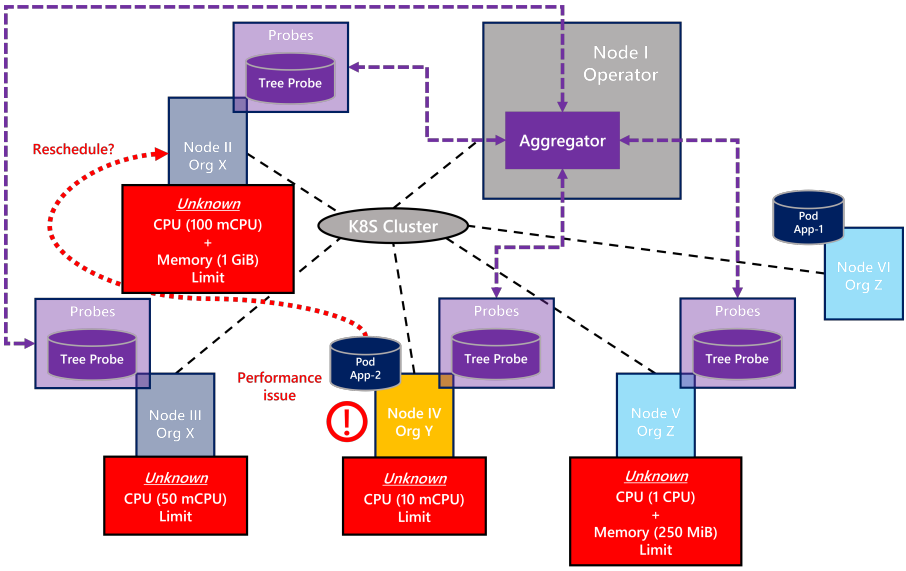


Fig. 4 An example use case to which the proposed probe swarm is applied.

and binding between pod and node takes place. This flow thus allows an operator to manually intervene a scheduling operation, based on the deployment of probes, which inform the operator about the potentially unknown underlying deployment cluster.

5 Evaluation

The case discussed in previous section can be simplified a bit, obtaining a situation as shown in Figure 4. It defines a cross-organizational collaboration between organizations X, Y, Z which is orchestrated by a central operator. An evaluation is presented in this section to illustrate the potential of the proposed probe mesh and to show to which performance improvements it could lead. The goal for the operator is to reschedule one of the dark blue collaboration pods. More specifically, the `App-2` pod, which is deployed at Organization Y initially, needs to be moved to another node, due to the reason explained in the next paragraph.

The Kubernetes cluster used for this evaluation consists of homogeneous nodes: each node is a virtual machine (VM) running Ubuntu 18.04 LTS and is equipped with four vCPUs of an Intel Xeon E5645 2.4 GHz processor and 4 GiB of RAM. There are no hardware differences between the nodes, and no additional background loads are deployed. To realize performance differences between nodes, local resource limits are set, as explained in Section 3. These limits, for now focusing on CPU and memory usage, are enforced by a hosting organization, and are thus unknown to the operator. This way, a heterogeneous cluster is realized, from the perspective of the operator. The local limits used in this example are shown in the Figure. Organization Y, hosting Node IV,

applies a restrictive CPU limit to the containers in the collaboration pod, as only one hundredth of a core may be consumed. This hurts performance significantly, implying the need for a rescheduling. The set of candidate nodes consists of Node II, III and V. Node VI is excluded from this set, as it already hosts the `App-1` pod, and for example an anti-affinity rule may require to put both pods on different nodes to achieve certain crash fault-tolerance.

It is assumed that the collaboration pods considered here represent a storage solution. A data structure probe, as suggested in Section 4, is thus selected by the operator for this evaluation scenario. Assume further that the performance of data lookups is crucial for the considered application. Therefore, a balanced BST tree may be selected as data structure type, as it is known for its relatively short lookup times. To obtain an idea of the relative performance differences between nodes and to obtain an idea of the time it takes to store a certain amount of data given the chosen data structure, it is sufficient to create, and to free afterwards, such a balanced BST tree and to perform time measurements. The following code fragment, written in C, shows how such a probing solution could be defined.

```
// Node representation
typedef struct _node {
    int id; char* data; struct _node* left, right;
} node;

// Probe part I
//   Create a balanced BST with 'max' nodes, each with 'bytes' data
void createTree(node** root, int min, int max, int bytes) {
    int id = (min + max) / 2;
    // Insert a leaf by following path from the root
    insertNode(root, id, bytes);
    if (min == max) { return; }
    if (id > min) { createTree(root, min, id - 1, bytes); }
    createTree(root, id + 1, max, bytes);
}

// Probe part II
//   Free dynamic memory occupied by all tree nodes
void freeTree(node* root);
```

Listing 2 A tree probe to capture the performance differences between nodes in the cluster.

The size of a tree node is equal to 32 bytes when the GCC compiler is used on a 64 bit machine: 4 bytes for the `ID` integer plus 4 padding bytes, 8 bytes for the data pointer, and 8 bytes for both the left and right child pointer. The `ID` field ranges from 1 to N and is used to sort the items in the BST, while the `data` field contains a string of D random alphanumerical characters. Using this code, the operator could launch multiple short-lived tree probes and experiment with different tree sizes, to obtain an overview of the performance differences between the nodes in the candidate set. A strategy could for example be to gradually intensify the probing experiment, by tuning the dominant parameters N and D, to prevent a probe from having a too significant impact on the performance of a node. The results of such an evaluation are presented in Table 1. The experiments are executed using containers with Ubuntu 18.04 image running a `sleep` process to keep the container alive. The

probe executable file and corresponding time measurements are then initiated using the following command:

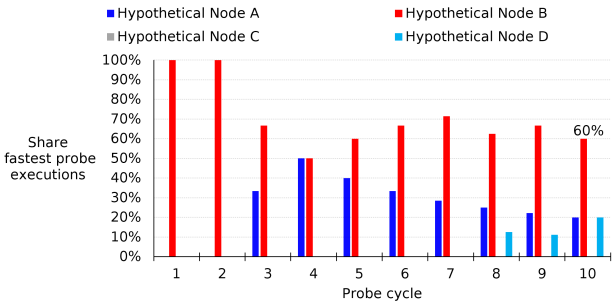
```
time -f %e ./tree.exe --type bst --balanced true --nodes N --data D
```

The table shows the sorted execution times of five probe cycles of the tree probe at the different candidate nodes for six configurations of N and the size of a single tree node $S = 32 + D$ in bytes. Only the configurations for which at least 100 MB needs to be allocated, based on the analysis of dynamic memory allocations, are evaluated. The reason for this is that smaller cases tend to only show negligible performance differences, while larger cases are too large for a 4 GiB RAM capacity. The obtained time measurements clearly match with the ratios of the local CPU limits: compared with reference Node IV, Node II executes about ten times quicker, Node III executes about five times quicker, and Node V executes about hundred times quicker. This result is to be expected from the evaluation setup presented here, but such an insight is important for the operator in an unknown setup. Significant performance increases may thus be gained when the `App-2` pod would be rescheduled. The deployment of the probe swarm furthermore shows that different candidate nodes may be recommended for the different probe configurations. This thus proves the need for probes, as it may be completely unclear for the operator which nodes will perform well under an unknown set of local settings. It becomes clear that some probe pods are not able to finish their execution, as their corresponding probe containers exceed the memory limits as defined by the local administrator of the hosting organization. This causes them to be terminated, more specifically to be out of memory (OOM) killed. The evaluation also confirms that parameter settings should be increased gradually, as suggested above. The probe executions for memory sizes larger than 100 MB may already take around one minute or more for the probe to finish. A possibility to solve this issue partially, is to stop a running probe at a Node X when its execution time significantly surpasses the one of an already finished probe at a Node Y. When no probe finishes quickly, it is needed to specify an upper limit on execution time. Finally, it is clear that when N increases, more overhead is present. The $N = 10^6$, $S = 32$ B + 1 kB configuration needs more time to execute and results in an extra OOM killed container, although the data structure allocates roughly the same amount of dynamic memory as in the $N = 10^4$, $S = 32$ B + 100 kB case.

The five probe measurements executed at each node, as presented in Table 1, are close to each other. This means that additional probe executions or cycles would not lead to significantly different conclusions, as no varying deployment conditions are considered. In a real scenario however, as already mentioned in step 9 of Section 4.2, it may be that dynamic factors such as background loads are present, causing probing results to vary and insights in the performance of nodes to change. It is therefore crucial that multiple probe cycles are evaluated over time. It is possible to illustrate this using a hypothetical scenario in which a probe swarm is deployed on four nodes and the results of ten probe cycles

Table 1 Time measurements in seconds of tree probe executions at Nodes II, III, IV and V for different configurations of N and S, which represent the number of tree nodes and size of a tree node in orders of bytes respectively.

	S: (32 + 10) B	S: 32 B + 1 kB	S: 32 B + 100 kB	S: 32 B + 1 MB
N: 10^2	4.2 kB ≪ RAM capacity	103.2 kB ≪ RAM capacity	10.0 MB ≪ RAM capacity	100.0 MB II: 3.9 4.1 4.1 4.2 4.3 s III: 8.0 8.1 8.2 8.2 8.3 s *IV: 41.1 41.4 41.5 41.7 42.1 s* → **V: 0.4 0.4 0.4 0.4 0.4 s**
N: 10^3	42 kB ≪ RAM capacity	1.0 MB ≪ RAM capacity	100.0 MB II: 4.3 4.3 4.4 4.6 4.7 s III: 8.4 8.4 8.5 8.5 8.7 s *IV: 42.5 42.7 42.8 43.1 44.5 s* → **V: 0.4 0.4 0.4 0.4 0.5 s**	1.0 GB → **II: 39.9 40.0 40.7 41.6 42.4 s** III: 80.4 80.8 81.8 82.1 84.2 s *IV: 414.4 417.2 418.6 428.6 429.7 s* V: OOM
N: 10^4	420 kB ≪ RAM capacity	10.3 MB ≪ RAM capacity	1.0 GB → **II: 42.3 42.4 42.6 42.9 43.3 s** III: 85.5 85.6 87.4 87.4 87.6 s *IV: 443.0 444.8 444.8 445.4 446.7 s* V: OOM	10.0 GB ≫ RAM capacity
N: 10^5	4.2 MB ≪ RAM capacity	103.2 MB II: 5.8 5.8 5.9 6.6 6.6 s III: 11.8 12.2 12.2 12.5 12.7 s *IV: 61.6 62.0 62.2 62.5 62.6 s* → **V: 0.6 0.6 0.6 0.6 0.7 s**	10.0 GB ≫ RAM capacity	100.0 GB ≫ RAM capacity
N: 10^6	42 MB ≪ RAM capacity	1.0 GB II: OOM → **III: 135.5 136.4 136.8 136.8 137.0 s** *IV: 728.9 735.0 737.6 739.2 740.2 s* V: OOM	100.0 GB ≫ RAM capacity	1.0 TB ≫ RAM capacity



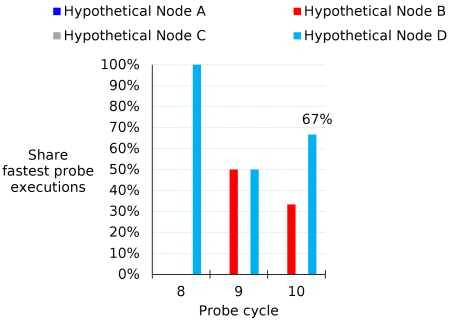
Example case I:
Weighted evaluation of *all cycles*

Node A fastest: cycles 3, 4 | **Node B fastest: cycles 1, 2, 5, 6, 7, 9** | Node C fastest: - | Node D fastest: cycles 8, 10

Fig. 5 The analysis of all probe cycles, suggesting the selection of Node B.

are gathered. The probe swarm measurements could and should be analyzed in multiple ways depending on the situation at hand. Figures 5, 6 and 7 show a possible evaluation under the assumption that it is needed to find the node which is most likely to perform best. This results in a binary evaluation per probe cycle, i.e. a node has the lowest probe execution time or not. Based on this evaluation, it is possible to calculate for each node its share in fastest probe executions. The three illustrations should be interpreted as follows:

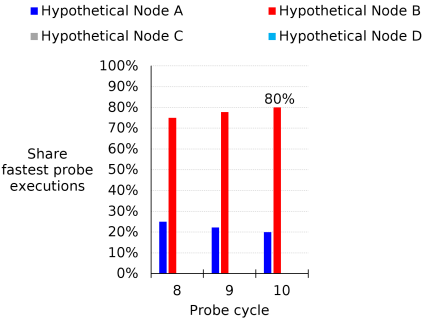
- Figure 5: The default scenario is that of a weighted evaluation of all cycles, for example using equal weights as shown here. Node C is not a candidate node as it never has the fastest probe execution. A reason could be that this node is lagging behind clearly, for example when a single probe execution takes longer than the time between two consecutive probe cycles, causing it to be excluded from further evaluation to speed up the evaluation process. Given the observations after ten cycles, the operator may decide to choose Node B as it has the highest chance of being the most performing one. This result depends on a number of parameters such as the duration of probing, the number of probing cycles and the weight distribution. The number of required probing cycles could depend on the expected duration of the application to be scheduled. When a longer-running job is considered, it would be better to consider more probe cycles, as it provides a more reliable historic view on node performance.
- Figure 6: Contrary, for shorter-running jobs, it is less valuable to take older probe cycles into account, as the more recent node performance measurements are more relevant. Only evaluating the latest three cycles is thus another possibility. The operator would then select Node D in this example, as this one shows promising results during the latest probe cycles.
- Figure 7: Additional criteria could be applied to the evaluation of the probe cycles. It could be chosen, for example, that a node at cycle x is considered fastest only if it lowers execution time with $> 30\%$ compared with the



Example case II:
Weighted evaluation of *latest three cycles*

Node A fastest: - | Node B fastest: cycle 9 | Node C fastest: - | **Node D fastest: cycles 8, 10**

Fig. 6 The analysis of the latest three probe cycles, suggesting the selection of Node D.



Example case III:
Weighted evaluation of *all cycles*, but with the extra condition that a node at cycle x is considered fastest only if it lowers execution time with >30% compared with the fastest node up until cycle x-1

Node A fastest: cycles 3, 4 | **Node B fastest: cycles 1, 2, 5, 6, 7, 8, 9, 10** | Node C fastest: - | Node D fastest: -

Fig. 7 The analysis of all probe cycles, but with an extra condition, again suggesting the selection of Node B.

fastest node up until cycle $x-1$. This way it is prevented that marginal performance changes have a significant impact on the decision of the operator. This would mean for the example that, although Node D suddenly shows promising results during the latest probe cycles, its performance results are only slightly better than those of Node B. It is therefore conceivable to just ignore them, causing the operator to be more confident about the selection of Node B.

This example makes clear that different data analyses are possible and that there is not necessarily a single correct solution. Which scheduling decision should be taken and which result it will bring depends on many factors. The operator could for instance follow the strategy to pick the node which is most

likely to perform best based on probe input. However, this may be an expensive node to schedule, in case costs are considered relevant in the collaboration. Furthermore, it may not necessarily be needed to pick the top-performing node. Imagine a video pipeline, for which the video processing probe is used. If it is only required to be able to process 25 frames per second, it is unnecessary to select any node for which probe evaluation shows a higher potential capacity, assuming the nodes considered show a comparable stability over time. The key message of this article is that probes are needed in a cross-organizational setup to fuel these types of analyses. Without them, an operator would only be able to perform limited analyses, and as such only gain limited insight in the performance differences between nodes in the cluster.

6 Conclusion

Probes are needed in a cross-organizational cluster to allow an operator to make ad hoc decisions on the placement of pods. There are simply too many uncertain factors from the perspective of this operator, as the worker nodes are managed by other organizations. Background loads may thus be present, the state of underlying hardware may be (partially) unknown, and local performance limits may be applied. These unique conditions in a dynamically composed cluster require the intervention of a probe swarm. As discussed, different types of probes are possible, ranging from generic probes to copy probes, and application-specific probes. The latter are perfectly suited to simulate performance differences between nodes when typical cross-organizational applications are considered. A possible integration of these probes into the vanilla Kubernetes scheduling pipeline is presented, allowing for proper rescheduling of misplaced pods. This rescheduling is especially important for the case at hand, as an ad hoc collaboration should not be delayed due to resource bottlenecks. Finally, a set of probes based on a BST are deployed and evaluated when local resource limits are applied to the container of the probe pod. It shows that these limits, which are unknown to the scheduling operator, can have a significant impact on the execution time and thus performance of the proposed application-specific probe: rescheduling the pod may improve performance with a factor five, ten or even hundred. Future work should focus on bottleneck discovery of microservice applications, being it cross-organizational services or not. There may be plenty of reasons for an application to underperform. Automatic mechanisms are thus required to trace application state, for example using a (distributed) tracing framework like OpenTelemetry [33]. This may be a complex task to solve when a multitude of services interact with each other. Based on gathered observations, it may be easier for an operator to pinpoint reasons behind an application stall. One of these reasons may be the misplacement of an application within the cluster, triggering the deployment of a probe swarm in the case of a cross-organizational setup.

Acknowledgments. The work described in this paper, was partly funded by the FUSE research project [2], in which a Flexible federated Unified Service

Environment was investigated. The project was realized in collaboration with imec. Industry project partners were Barco, Axians and e-BO Enterprises, with project support from VLAIO (Flanders Innovation & Entrepreneurship).

References

[1] Kubernetes. https://kubernetes.io. Accessed March 1 2022.

[2] FUSE: Flexible federated Unified Service Environment. https://www.imec-int.com/en/what-we-offer/research-portfolio/fuse. Accessed March 1 2022

[3] Goethals, T., Kerkhove, D., Van Hoye, L., Sebrechts, M., De Turck, F., Volckaert, B.: Fuse: A microservice approach to cross-domain federation using docker containers. In: Proceedings - 9th International Conference on Cloud Computing and Services Science (CLOSER), pp. 90–99. SciTePress, Heraklion, Greece (2019). https://doi.org/10.5220/0007706000900099

[4] Wickboldt, J.A., Guerreiro, M.Q., Granville, L.Z., Gaspary, L.P., Schwarz, M.F., Guok, C., Chaniotakis, V., Lake, A., MacAuley, J.: Meican: Simplifying dcn life-cycle management from end-user and operator perspectives in inter-domain environments. IEEE Communications Magazine **56**(1), 179–187 (2018). https://doi.org/10.1109/MCOM.2017.1601205

[5] Van Hoye, L., Wauters, T., De Turck, F., Volckaert, B.: Trustful ad hoc cross-organizational data exchanges based on the hyperledger fabric framework. Int J Network Mgmt **30**(6), 2131 (2020). https://doi.org/10.1002/nem.2131

[6] Van Hoye, L., Wauters, T., De Turck, F., Volckaert, B.: A secure cross-organizational container deployment approach to enable ad hoc collaborations. Int J Network Mgmt, 2194 (2021). https://doi.org/10.1002/nem.2194

[7] Medel, V., Tolón, C., Arronategui, U., Tolosana-Calasanz, R., Bañares, J.Á., Rana, O.F.: Client-side scheduling based on application characterization on kubernetes. In: Proceedings - 14th International Conference on Economics of Grids, Clouds, Systems and Services (GECON), pp. 162–176. Springer, Biarritz, France (2017). https://doi.org/10.1007/978-3-319-68066-8_13

[8] Rattihalli, G., Govindaraju, M., Lu, H., Tiwari, D.: Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes. In: Proceedings - IEEE 12th International Conference on Cloud Computing (CLOUD), pp. 33–40. IEEE, Milan, Italy (2019). https://doi.org/10.1109/CLOUD.2019.00018

[9] Zhong, Z., Buyya, R.: A cost-efficient container orchestration strategy in kubernetes-based cloud computing infrastructures with heterogeneous resources. ACM Trans. Internet Technol. **20**(2), 1–24 (2020). https://doi.org/10.1145/3378447

[10] Santos, J., Wauters, T., Volckaert, B., De Turck, F.: Resource provisioning in fog computing: From theory to practice. Sensors **19**(10), 2238 (2019). https://doi.org/10.3390/s19102238

[11] Santos, J., Wauters, T., Volckaert, B., De Turck, F.: Towards delay-aware container-based service function chaining in fog computing. In: Proceedings - IEEE/IFIP Network Operations and Management Symposium (NOMS), pp. 1–9. IEEE, Budapest, Hungary (2020). https://doi.org/10.1109/NOMS47738.2020.9110376

[12] Rossi, F., Cardellini, V., Lo Presti, F., Nardelli, M.: Geo-distributed efficient deployment of containers with kubernetes. Computer Communications **159**, 161–174 (2020). https://doi.org/10.1016/j.comcom.2020.04.061

[13] Townend, P., Clement, S., Burdett, D., Yang, R., Shaw, J., Slater, B., Xu, J.: Invited paper: Improving data center efficiency through holistic scheduling in kubernetes. In: Proceedings - IEEE International Conference on Service-Oriented System Engineering (SOSE), pp. 156–15610. IEEE, San Francisco, CA, USA (2019). https://doi.org/10.1109/SOSE.2019.00030

[14] Rocha, I., Göttel, C., Felber, P., Pasin, M., Rouvoy, R., Schiavoni, V.: Heats: Heterogeneity-and energy-aware task-based scheduling. In: Proceedings - 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 400–405. IEEE, Pavia, Italy (2019). https://doi.org/10.1109/EMPDP.2019.8671554

[15] Topology Aware Scheduling. https://github.com/kubernetes-sigs/scheduler-plugins/tree/master/kep/119-node-resource-topology-aware-scheduling. Accessed March 1 2022.

[16] El Haj Ahmed, G., Gil-Castiñeira, F., Costa-Montenegro, E.: Kubcg: A dynamic kubernetes scheduler for heterogeneous clusters. Software: Practice and Experience **51**(2), 213–234 (2021). https://doi.org/10.1002/spe.2898

[17] Vaucher, S., Pires, R., Felber, P., Pasin, M., Schiavoni, V., Fetzer, C.: Sgx-aware container orchestration for heterogeneous clusters. In: Proceedings - IEEE 38th International Conference on Distributed Computing Systems (ICDCS), pp. 730–741. IEEE, Vienna, Austria (2018). https://doi.org/10.1109/ICDCS.2018.00076

[18] KEP - Trimaran: Real Load Aware Scheduling. https://github.com/kubernetes-sigs/scheduler-plugins/tree/master/kep/61-Trimaran-real-load-aware-scheduling. Accessed March 1 2022.

[19] Tzenetopoulos, A., Masouros, D., Xydis, S., Soudris, D.: Interference-aware orchestration in kubernetes. In: Proceedings - International Conference on High Performance Computing, pp. 321–330. Springer, Frankfurt, Germany (2020). https://doi.org/10.1007/978-3-030-59851-8_21

[20] Menouer, T.: Kcss: Kubernetes container scheduling strategy. J Supercomput **77**, 4267–4293 (2021). https://doi.org/10.1007/s11227-020-03427-3

[21] Rausch, T., Rashed, A., Dustdar, S.: Optimized container scheduling for data-intensive serverless edge computing. Future Generation Computer Systems **114**, 259–271 (2021). https://doi.org/10.1016/j.future.2020.07.017

[22] Delgado, P., Didona, D., Dinu, F., Zwaenepoel, W.: Job-aware scheduling in eagle: Divide and stick to your probes. In: Proceedings - Seventh ACM Symposium on Cloud Computing, pp. 497–509. Association for Computing Machinery, Santa Clara, CA, USA (2016). https://doi.org/10.1145/2987550.2987563

[23] Bayer, T., Moedel, L., Reich, C.: A fog-cloud computing infrastructure for condition monitoring and distributing industry 4.0 services. In: Proceedings - 9th International Conference on Cloud Computing and Services Science (CLOSER), pp. 233–240. SciTePress, Heraklion, Greece (2019). https://doi.org/10.5220/0007584802330240

[24] Kubernetes 1.19.16 - noderesources/fit.go. https://github.com/kubernetes/kubernetes/blob/v1.19.16/pkg/scheduler/framework/plugins/noderesources/fit.go#L230. Accessed March 1 2022.

[25] Nodes - Resource capacity tracking. https://kubernetes.io/docs/concepts/architecture/nodes/#node-capacity. Accessed March 1 2022.

[26] Reserve Compute Resources for System Daemons - Node Allocatable. https://kubernetes.io/docs/tasks/administer-cluster/reserve-compute-resources/#node-allocatable. Accessed March 1 2022.

[27] Nodes - Info. https://kubernetes.io/docs/concepts/architecture/nodes/#info. Accessed March 1 2022.

[28] Docker - Runtime options with Memory, CPUs, and GPUs. https://docs.docker.com/config/containers/resource_constraints. Accessed March 1 2022.

[29] Hashman, E.: New in Kubernetes v1.22: alpha support for using swap memory. https://kubernetes.io/blog/2021/08/09/run-nodes-with-swap-alpha (2021). Accessed March 1 2022.

[30] Bellard, F.: TachusPI Documentation. https://bellard.org/pi/pi2700e9/readme.html (2009). Accessed March 1 2022.

[31] Scheduler Configuration. https://kubernetes.io/docs/reference/scheduling/config. Accessed March 1 2022.

[32] Scheduling Framework. https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework. Accessed March 1 2022.

[33] OpenTelemetry. https://opentelemetry.io. Accessed March 1 2022.

Laurens Van Hoye is a PhD student in the Internet Technology and Data Science Lab (IDLab) of Ghent University, Belgium and imec. He graduated as a Master of Science in Computer Science Engineering in 2017 from the same university. His PhD, which is supervised by Prof. Filip De Turck and Prof. Bruno Volckaert, focuses on cloud-based research topics, such as distributed and decentralized systems, containerization and orchestration, and authentication and authorization services.

Tim Wauters obtained his M.Sc. and PhD degrees in electro-technical engineering from Ghent University in 2001 and 2007 respectively. He has been working as a post-doctoral fellow of the F.W.O.-V. in the Department of Information Technology (INTEC) at Ghent University, and is now also active as a senior researcher at imec. His main research interests focus on design and management of networked services, covering multimedia distribution, cybersecurity, big data and smart cities. His work has been published in more than 150 scientific publications.

Bruno Volckaert is professor cloud and advanced software engineering in the Department of Information Technology (INTEC) at Ghent University in collaboration with imec. His current research deals with scalable, reliable and high performance distributed software systems for a.o. Smart Cities and industry4.0, SOLID, scalable cybersecurity detection and mitigation architectures and autonomous optimization of cloud-edge-fog-based applications. He is author or co-author of more than 180 peer-reviewed papers published in international journals and conference proceedings.

Filip De Turck leads the Network and Service Management Research Group, Department of Information Technology, Ghent University–imec, Belgium. He (co) authored over 450 peer reviewed articles. His research interests include network and service management and the design of efficient virtualized networks. He serves as the Chair for the IEEE Technical Committee on Network

Operations and Management (CNOM) and a TPC for many network and service management conferences.