New solutions for exposing clustered applications deployed in the cloud

Sebastian Łaskawiec¹ · Michał Choraś¹ · Rafał Kozik¹

Received: 2 October 2017 / Revised: 2 January 2018 / Accepted: 6 March 2018 / Published online: 12 October 2018
© The Author(s) 2018

Abstract

Hosting applications in the cloud has become very popular. In this context, automatic application scaling, health checking and bootstrapping required services has never been easier. However, there are some types of services that require direct communication between a server and a client, bypassing the cloud. Popular examples include data grids (distributed caching solutions) and gaming servers. Most of these services use custom network protocols based on TCP or UDP transport, along with client-side load balancing, for optimizing network traffic. Unfortunately, most cloud vendors use HTTP-based reverse proxies and are unable to route custom protocols effectively. The major contribution of this paper is a proposed method for using a single load balancer per application instance (replica) to solve the problem of effective communication between a clustered application deployed within the cloud, and a client hosted in the outside world.

Keywords Cloud · Containers · Load balancer · Kubernetes

1 Introduction and problem statement

Components offered by modern cloud vendors are often designed for standard web applications. In most cases, such an application consists of backend services, which implement business logic; a frontend web server, which communicates with users and a data store—typically a relational database. When an application hosts content for multiple users, it is often recommended (or even required) to use a caching layer between the application backend and the data store layer. Caching is also a good fit for modern application architectures, such as microservices [1] where a single user request might cascade into multiple calls between backend services. Each call introduces additional latency and decreases the overall user experience.

Modern caching solutions, often known as Data Grids, need to respond to high application demands and offer data storage within a distributed system without a single master node. This goal can be achieved by using consistent hashing for data partitioning which allows data access with

$O(1)$ complexity [2]. Consistent hashing can also be used by a remote client application for calculating which node in a data grid cluster owns the data, and how to access the data within the shortest possible path (without redirecting from one node to another).

Each system needs to be deployed on appropriate hardware in order to provide value to users. Each system may have different demands, and IT departments inside companies might focus on different characteristics (e.g. time to market, or time required to onboard a new development crew member). Cloud computing fits perfectly into those requirements and has therefore gained popularity very quickly [3]. Enterprise application demands very often require hybrid approach between a private cloud (where all components are installed in a standard on-premise data center) and a public cloud (where all components are hosted by a cloud vendor). Those scenarios are especially challenging from a discovery and connectivity point of view. Some cloud platforms (such as Kubernetes [4] or OpenShift [5]) encapsulate all the traffic within the cloud, exposing a single point for accessing inner components (Fig. 1). Since most vendors aim for a standard web application which uses the HTTP(S) protocol for communication with the client, the component responsible for ingress traffic is usually a reverse proxy rather than an edge

✉ Sebastian Łaskawiec
sebastian.laskawiec@gmail.com

¹ Telecommunications and Computer Science, University of Science and Technology UTP, Bydgoszcz, Poland

Fig. 1 Accessing application instances between clouds

router or L4 (Transport Layer of OSI model) load balancer. A reverse proxy usually offers more features related to the HTTP protocol (such as load balancing and content compression) than an edge router (which is focused on L3/Network routing) or L4 load balancer (which uses transport layer information to make balancing decisions). This approach is very convenient from a security point of view where a single component is responsible for both defending the system from unwanted traffic (the firewall) and handling encryption/decryption (SSL termination).

Although this architecture makes a lot of sense for a typical application, some types of deployments need to consume topology information in order to optimize system performance. Common examples are high efficiency data grids and gaming systems. Many video games use RTP (Real-time Transport Protocol), which was originally designed for delivering audio and video content, for communicating with one or more dedicated servers. Those systems can be viewed as if they were using a client-based load balancing technique [6] where topology information is essential to fulfill all application requirements. Apart from caching and gaming, there are also a number of hybrid cloud applications where part of the cluster is hosted inside a public cloud and part of it is hosted within a traditional on-premises data center. This approach can sometimes be seen when using quorum-based systems where a majority of instances are hosted in the cloud for stability, while a minority are hosted within a data center to improve communication speed.

Enabling client-based load balancing requires achieving two major properties—discoverability and connectivity. Discoverability represents a way to identify topology information—in particular the number of server instances, exposed ports and IP addresses. This enables the client to configure the connection pool according to that information. Connectivity represents an ability to connect to those internal instances directly. It is also worth mentioning that both the caching and gaming industry have very high

demands on application throughput at low latency. Therefore, an ideal solution needs to have minimal negative impact on both of those characteristics.

The major contributions of this paper are:

- The proposal and evaluation of solutions for exposing services hosted in the cloud to the outside world
- A method for enabling client-side routing using a load balancer per application instance for a client application is hosted outside of the cloud
- A discussion on benchmark results followed by a general recommendation for using different communication flows depending on the use case

This paper is structured as follows: Sect. 2 describes related work both conceptually and technically; Sect. 3 presents a number of proposed solutions and describes a set of requirements for evaluating a prototype; Sect. 4 evaluates a prototype selected from Sect. 3 according to the requirements and provides some implementation details as well as benchmarks; and Sect. 5 contains a discussion of results and planned future work.

2 Related work

Methods for exposing an application cluster deployed in the cloud to the outside world have been investigated since the early days of cloud computing. The “Cluster as a Service” [7] approach based on RVWS (Resources Via Web Service) framework published in 2010 proposes an abstraction to expose a set of WSDL-based web services deployed on multiple cloud vendors. One of the most important characteristics of the framework is the ability to discover new clusters automatically. This is made possible by using a broker architecture which connects all exposed clusters. A “client” application along with a “cloud provider” specifies a required SLA (Service Level Agreement) for the client’s request or work flow (Fig. 2). An SLA is a

Fig. 2 Cluster as a Service overview [7]

type of contract between a cloud provider and a client that describes the performance and functional characteristics of a service.

The RVWS framework uses a Publisher Service to expose cluster characteristics in WSDL format. Based on this information, the Broker can decide which cluster best meets client requirements (Fig. 3).

Even though the “CaaS” framework aims at different goals than those of this paper, it relevantly solves both the

discovery and connectivity problems within the cloud. The RVWS framework, which is a fundamental part of the system, focuses on batch processing tasks and WSDL-based web services whereas this paper focuses on high performance and low latency communication required by modern caching solutions and the gaming industry. To address this, solutions presented in this paper utilize basic parts of a “Publisher Service” and offer direct communication with the cluster (similarly to “CaaS”). Since both solutions allow the client application to communicate directly with the cluster, the security concerns remain similar. However, WSDL-based solutions are often based on the HTTP protocol rather than TCP connection with custom binary protocols (network protocols which use binary arrays to encode commands and data, often used in caching solutions). Since both edge routers and reverse proxies offer many more configuration options for HTTP, we should consider “CaaS” to be more secure than solutions proposed in this paper.

Exposing a cluster within the cloud is also similar (to some extent) to multi-tenancy [8]. The distinction is that multi-tenancy is based on tenant recognition, whereas exposing a cluster requires recognizing a specific server (to which a specific request should be redirected). For HTTP-based protocols there are number of options including cookie-based routing, or using a Host HTTP header. However, high performance protocols are often limited to binary data based on TCP or UDP transport. Even though the TCP protocol itself does not have any support for these

Fig. 3 Exposing a cluster using Publisher Service [7]

kind of solutions, encrypting traffic using Transport Layer Security (TLS) with Server Name Indication (SNI) allows the use of the `HostName` field [9], which contains a fully qualified server name. This approach has been successfully used for the multi-tenancy implementation shown in Fig. 4.

An interesting approach (analyzed in depth in the following paragraphs) is to use a load balancer for each server deployed in the cloud, since most of the cloud vendors use high performance solutions for load balancing. A similar type of functionality has been provided by Fabric8's Expose Controller [10]. This controller allows exposure of a load balancer per group of server instances (not a load balancer per server) and it does not provide any information about internal/external address mapping. Still, from a technical perspective, the Expose Controller was used as a source of inspiration for the technical solution presented in this paper.

Finally, all application instances need to operate on some sort of hardware. The scheduler is typically responsible for assigning an application replica to a node that runs the program. Even though scheduling algorithms are not strictly connected to the proposal mentioned in this paper, it is worth to mention that it is an interesting and very important area of research for cloud-related applications [11–14].

3 The proposed solution

Solutions for exposing clusters deployed within the cloud to the outside world need to meet discoverability and connectivity criteria. A client application needs to be able to discover nodes in the cluster, as well as to establish connections to individual cluster members. A good solution also needs to introduce the smallest possible penalty on performance (defined by throughput in operations per second). The final goal is full process automation which is required from a cluster auto healing perspective. A full list of requirements and priorities is presented in Table 1.

One of the solutions initially evaluated was securing the transport with TLS and use of the SNI extension to decide which server receives the request from the client (Fig. 4).

Fig. 4 Multi-tenancy based on SNI routing

This solution is natively supported by many reverse proxy applications used by cloud vendors (such as HAProxy [15] or NGINX [16]) thus it can be easily automated. The biggest downside of this approach is that the performance is 30% lower (Table 2) than using non-TLS encrypted transport.

An alternative approach is to use a custom load balancer application which is able to decode additional, non-standard TCP and UDP options (as many data grid and gaming industry applications use TCP or UDP as transport protocols). However, during the evaluation it became clear that only a small fraction of libraries allow the use of custom options on the socket level, one of the commonly used examples being JNetPcap [17]. With this in mind, the router solution was discarded as only a small proportion of projects would be able to leverage this approach.

The next group of possible solutions are those based on the use of load balancers provided by the cloud vendors. For many cloud vendors, load balancers and reverse proxies are considered one and the same, however, they are considered separately in this paper. Most solutions of this nature (such as Amazon Elastic Load Balancer [18] and Google Cloud Load Balancer [19]) are optimized to serve high volume traffic whilst introducing the smallest possible loss of performance. Network Virtualization efforts which integrate seamlessly with both load balancers and SDN are used within the cloud to achieve much better results than ever before [19, 20]. To solve the connectivity problem, it is possible to allocate a load balancer per node in the cluster. Since creating a new load balancer and a new binary proxy very often requires a REST API call to an API exposed by a cloud vendor, this process can be easily automated. Finally, the discoverability problem can be solved by creating a new application which exposes mappings between internal and external IP addresses for a cluster deployed in a cloud. Therefore, our final proposition is presented in Fig. 5.

Each load balancer instance has a publicly accessible IP address, and in fact, many cloud vendors treat load balancers as an external/virtual IP implementation [21]. Each request is passed into a binary proxy (a proxy which forwards messages sent using binary protocols), then finally to an application instance. The motivation behind implementing proxying is that in container-based clouds, application instances can often be restarted (e.g. because of balancing hardware resources utilization or auto-scaling in and out process). The proxy acts as a safety buffer pausing any communication until application instances are ready to process incoming requests. Another argument is that creating and destroying load balancer instances can take some time (on Google Cloud Platform it is a matter of minutes).

Table 1 Requirements for proposed solution

Solution	Priority
Discoverability	Must-have
Connectivity	Must-have
Possibility of automation	Good-to-have
Small performance penalty	Good-to-have

Table 2 Performing 10,000 Cache PUT operations with different TLS configuration [8]

Test description	Average result	Units
Single server without TLS	430.075	ms/test
Single server with TLS/SNI	847.909	ms/test

With binary proxies, the number of create/destroy events can be limited.

Use of load balancers and binary proxies satisfy the connectivity requirement. A newly created component called an External IP Controller instance is responsible for automating the process of discovering new application instances and creating proxies, as well load balancers for all instances. This component also exposes a REST endpoint (which serves YAML-based content with internal/external IP address mapping) that helps the client application to determine which application instances are mapped to which addresses. This thereby satisfies the discoverability requirement. However, during application benchmarking it turned out that implementing binary proxy communication degrades the performance (Table 3).

Therefore it was decided to remove binary proxy (along with all its benefits) and propose the final optimized prototype without it.

4 Prototype evaluation

The final prototype was implemented using open source technologies such as Kubernetes. It will be implemented as cloud platform hosted on Google Cloud Engine and Infinispan for testing client/server communication. The main argument for choosing Kubernetes is that it does not depend on any particular cloud vendor (in other words it can be hosted using any public cloud provider such as Google Cloud Platform or Amazon Web Services). As for client/server communication, Infinispan is a data grid implementation which uses a Consistent Hash algorithm for locating primary owner [2] nodes for a specific portion of the data. This optimization tactic allows us to assess whether the prototype improves communication performance or not. The evaluated systems has been shown in Fig. 6.

Hot Rod is the name of Infinispan's custom binary protocol used for transmitting data to/from the Infinispan server. The Hot Rod client obtains topology information upon its first connection to the cluster. The topology contains a list of servers and their internal addresses (10.0.5.5, 10.0.6.5 and 10.0.7.5). The External IP Controller service provides a mapping (using a REST service) between internal (e.g. 10.0.5.5) and corresponding external addresses (e.g. 104.155.17.202). This information allows the client application to reconstruct the a consistent hash provided by the Infinispan server. Each topology update follows the same update pattern on the client side. The biggest

Fig. 5 Our system proposition

Table 3 Binary proxy benchmark results

Test description	With binary proxy	Average value	Units
Perform 10,000 Put Operations	Yes	352.822	ms/op
Perform 10,000 Put and Get Operations	Yes	131.926	ms/op
Perform 10,000 Put Operations	No	131.926	ms/op
Perform 10,000 Put and Get Operations	No	221.711	ms/op

Fig. 6 The diagram of the evaluated system

```

Input: External address
Output: Internal address

externalAddress = internalAddress
if (needsExternalMapping())
    externalAddress = getExternalAddressFromCache(internalAddress)
    if (externalAddress == null)
        externalAddress = queryExternalIpController(internalAddress)
        putInExternalAddressCache(internalAddress, externalAddress)
return externalAddress

```

Fig. 7 Hot Rod internal/external address mapping algorithm

advantage of this approach is that modification only needs to be applied to the client code¹, and the server is not aware of any translation process.

The algorithm used by the Hot Rod client is presented in Fig. 7.

The creation and removal of load balancers has been automated within an External IP Controller. The algorithm used by the controller has been designed to run indefinitely. The application thread wakes up every 5 minutes to query the Kubernetes API for all application instances. To create a Load Balancer for each application instance, the controller needs to set marker labels on each instance. Later, Kubernetes Services selects proper instances based on a Selector, where the Selector query and marker labels need to match. Figure 8 presents the algorithm in pseudo-code notation.

¹ The code can be accessed using the following URL <https://github.com/infinispan/infinispan/pull/5164>

```

while (true)
    applicationInstances = queryPlatformForInstances
    for (applicationInstance : applicationInstances)
        addMarkerLabels(applicationInstance)
        ensureLoadBalancerIsRunning(applicationInstance)
    removeUnnecessaryLoadBalancers()
    sleep(5 min)

```

Fig. 8 External IP Controller algorithm

System benchmarks were established using topology-aware and a simple Hot Rod client. The topology-aware client uses consistent hash information for sending each request to specific server, whereas the simple client chooses random server from specified connection list. Each test consists of sending 10,000 Put Operations (which means inserting 10,000 random strings into the data grid) and 10,000 Put and Get Operations (where the Get Operation represents getting a value previously inserted by a Put Operation). Each test was ran 31 times and an error margin was calculated using a 99.9% confidence interval. The results have been shown in Table 4 and 5 as well as in Figs. 9 and 10.

To check whether benchmark results differed from each other in any meaningful way, a test of significance was performed for each pair of results in Table 6. The

Table 4 Benchmark results for performing 1000 Put operations

Test description	With binary proxy	Average value	Error	Units
Topology aware client inside Kubernetes with Kubernetes internal addresses	L3 + Kubernetes internal	674.244	16.654	ms/ op
Topology aware client inside Kubernetes with Kubernetes Load balancers	L3 + Kubernetes internal + LB	779.557	87.575	ms/ op
Simple client inside Kubernetes with Kubernetes Load balancers	L1 + Kubernetes internal + LB	1072.407	36.502	ms/ op
Topology aware client inside the same data center with Kubernetes Load balancers	L3 + Kubernetes external + LB	1052.891	31.218	ms/ op
Simple client inside the same data center with Kubernetes Load balancers	L1 + Kubernetes external + LB	1465.567	176.349	ms/ op

Table 5 Benchmark results for performing 1000 Put and Get operations

Test description	Short description	Average value	Error	Units
Topology aware client inside Kubernetes with Kubernetes internal addresses	L3 + Kubernetes internal	1288.437	136.207	ms/ op
Topology aware client inside Kubernetes with Kubernetes Load balancers	L3 + Kubernetes internal + LB	1461.510	64.33	ms/ op
Simple client inside Kubernetes with Kubernetes Load balancers	L1 + Kubernetes internal + LB	2163.873	141.332	ms/ op
Topology aware client inside the same data center with Kubernetes Load balancers	L3 + Kubernetes external + LB	2465.586	85.034	ms/ op
Simple client inside the same data center with Kubernetes Load balancers	L1 + Kubernetes external + LB	2684.984	114.993	ms/ op

Fig. 9 Benchmark results for performing 1000 Put and Get operations

test is based on a null hypothesis which assumes that both tested mean values are equal (see equation below). The research hypothesis is that the means are different [22].

$$\mu_0 = \mu_1 \quad (1)$$

Since the samples are independent from each other, and measured groups are not linked to each other, the

Fig. 10 Benchmark results for performing 1000 Put and Get operations

Table 6 Significance measure matrix between benchmark results

	L3 + Kubernetes internal	L3 + Kubernetes internal + LB	L1 + Kubernetes internal + LB	L3 + Kubernetes external + LB
L3 + Kubernetes internal + LB	−4.307			
L1 + Kubernetes internal + LB	−36.182	−11.254		
L3 + Kubernetes external + LB	−39.018	−10.719	1.481	
L1 + Kubernetes external + LB	−16.288	−12.703	−7.96	31.4

significance test can be done using the “Two-Sample t-Test for Equal Means”:

$$Z = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}} \quad (2)$$

where $\bar{X}_1$ and $\bar{X}_2$ are sample means, σ_1^2 and σ_2^2 are variances and n_1 and n_2 are sample sizes.

The final step is to prove that the calculated Z value is between $(-\infty, 1 - \frac{\alpha}{2}) \cup (1 - \frac{\alpha}{2}, \infty)$ for the tested hypothesis that $\mu_0 \neq \mu_1$. For $\alpha = 0.05$ (99%), the tested interval is equal to $(-\infty, -2.58) \cup (2.58, \infty)$. Table 6 contains the calculated Z parameter for each test pair. We rejected the null hypothesis for each tested pair except “L1 + Kubernetes internal + LB” and “L3 + Kubernetes external + LB”. In that case, differences between values were not statistically significant.

5 Conclusion and future work

Exposing services hosted within the cloud to external consumption is not generally very popular, yet it is very useful for some use cases. Many cloud vendors have

sophisticated tools or templates that allow them to bootstrap such services very quickly. Automatic auto scaling based on CPU and memory metrics, health monitoring, and automatic backup management are only a few reasons why hosting applications within the cloud is preferred by many development teams. Many cloud vendors use load balancers as the only way of reaching services hosted within the cloud from the outside world. Very frequently the process of provisioning load balancers is strictly connected to creating new firewall or port forwarding rules. The process of creating all the necessary components is often automated and takes no longer than few minutes. The most crucial downside of using load balancers is that they are relatively expensive and very often charged on hourly basis.

The network infrastructure maintained by cloud vendors is very often optimized for using load balancers. Comparing “L3 + Kubernetes internal” and “L3 + Kubernetes internal + LB” benchmark results from the Table 5 showed only 13.43% lower throughput when sending data through a load balancer. The binary proxy implementation presented in paragraph 3 achieved much worse results. Comparing “Perform 10,000 Put and Get Operations” with

and without binary proxy from the Table 3, resulted in 68.05% worse performance.

The communication performance degrades much further when using other Virtual Machines within the same cloud offering (but not in the same container cloud). An example in the benchmark used a Kubernetes cluster running on a CentOS VM and a client application deployed in a separate VM instance. Comparing benchmarks for a topology aware client (“L3 + Kubernetes external + LB” and “L3 + Kubernetes internal + LB” benchmark results from the Table 5) showed 68.70% performance loss when the client is deployed in a separate VM instance. Benchmarks for a client that doesn’t use consistent hash-based routing the results were much better and stabilized at 24.08% worse performance when the client is deployed in a separate VM instance. The figures might significantly vary from cloud vendor to cloud vendor, due to network optimizations used in the data center.

A general recommendation based on the results is that one should always use direct communication within a container-based cloud when possible. This approach presented the best performance results among all tested configurations. Using the consistent hash-based routing is always the fastest approach from the client perspective since it picks the shortest possible path to a proper server. If for some reason this approach is not feasible (e.g. configured network policies do not allow direct communication), one should use a load balancer per instance approach as presented in this paper. This scenario resulted in 13.43% performance drop opposed to direct communication. In many cases such small performance penalty is perfectly acceptable. Another group of use cases are client applications deployed outside of the cloud that connects to a service deployed inside of the cloud. Using a load balancer per instance approach presented in this paper enables the client to pick the destination address (e.g. by using a consistent hash). Comparing “L3 + Kubernetes external + LB” and “L1 + Kubernetes external + LB” benchmark results from the Table 5 showed that using topology information resulted in 8.90% better performance results than sending requests to randomly-picked instances. Even though 8.90% might not seem much it is worth mentioning that the benchmark used relatively small payloads—random 16-character strings. Sending such small portions of data over an optimized project Andromeda network stack between data grid instances is relatively fast. The difference between those benchmarks is expected to be greater when using payloads of several megabyte size.

It is also worth to mention that adding TLS encryption resulted in 30% worse performance. High performance penalty was a deciding factor to skip security aspect from this paper.

Even though benchmark results look promising, there is still a need to benchmark other scenarios (e.g. involving disk access) and different types of services. Furthermore, this paper did not investigate security aspects of the proposed approach. Exposing internal to external (and vice versa) address mapping might be used as a potential attack vector. Some existing implementations like Infinispan may mitigate this risk by putting address mapping into the server. This way the client obtains an already-altered list of servers and is not aware of any mapping process.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Łaskawiec, S.: The evolution of java based software architectures. *J. Cloud Comput. Res.* **2**(1), 1–17 (2016)
2. Decandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Voshall, P., Vogels, W.: Dynamo: amazon’s highly available key-value store. *SIGOPS Oper. Syst. Rev.* **41**(6), 205–220 (2007)
3. Gartner Says Worldwide Public Cloud Services Market to Grow 18 Percent in 2017. <http://www.gartner.com/newsroom/id/3616417>
4. Kubernetes | Production-Grade Container Orchestration. URL: <https://kubernetes.io/>
5. OpenShift: Container Application Platform by Red Hat, Built on Docker and Kubernetes. <https://www.openshift.com/>
6. Cardellini, V., Colajanni, M., Yu, P.S.: Dynamic load balancing on web-server systems. *IEEE Internet Comput.* **3**, 28–39 (1999). <https://doi.org/10.1109/4236.769420>
7. Goscinski, A., Brock, M.: A technology to expose a cluster as a service in a cloud. In: Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing, vol. 107, pp. 3–12 (2010). <http://dl.acm.org/citation.cfm?id=1862295>
8. Łaskawiec, S., Choraś, M.: Considering service name indication for multi-tenancy routing in cloud environments. In: Advances in Intelligent Systems and Computing, vol. 525, pp. 271–278 (2017)
9. Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J., Wright, T.: Transport Layer Security (TLS) Extensions (2003). <https://tools.ietf.org/html/rfc6066>
10. Additional kubernetes controllers from fabric8 you can use with your microservice. <https://blog.fabric8.io/additional-kubernetes-controllers-from-fabric8-you-can-use-with-your-microservice-3126a2c4c132>
11. Sfrent, A., Pop, F.: Asymptotic scheduling for many task computing in Big Data platforms. *Inf. Sci.* **319**(Supplement C), 71–91 (2015). <https://doi.org/10.1016/j.ins.2015.03.053> (Energy Efficient Data, Services and Memory Management in Big Data Information Systems)
12. Sirbu, A., Pop, C., Șerbănescu, C., Pop, F.: Predicting provisioning and booting times in a metal-as-a-service system. *Future Gener. Comput. Syst.* **72**(Supplement C), 180–192 (2017). <https://doi.org/10.1016/j.future.2016.07.001>

13. Vasile, M.A., Pop, F., Tutueanu, R.I., Cristea, V., Kołodziej, J.: Resource-aware hybrid scheduling algorithm in heterogeneous distributed computing. *Future Gener. Comput. Syst.* **51**(Supplement C), 61–71 (2015) (**Special Section: A Note on New Trends in Data-Aware Scheduling and Resource Provisioning in Modern HPC Systems**)
14. Bessis, N., Sotiriadis, S., Pop, F., Cristea, V.: Using a novel message-exchanging optimization (MEO) model to reduce energy consumption in distributed systems, using a novel message-exchanging optimization (MEO) model to reduce energy consumption in distributed systems. *Simul. Model. Pract. Theory* **39**(Supplement C), 104–120 (2013) (**S.I. Energy efficiency in grids and clouds**)
15. HAProxy—The Reliable, High Performance TCP/HTTP Load Balancer. <http://www.haproxy.org/>
16. nginx. <https://nginx.org/en/>
17. jNetPcap OpenSource DPI SDK. <http://jnetpcap.com/>
18. AWS | Elastic Load Balancing—Cloud Network Load Balancer. <https://aws.amazon.com/elasticloadbalancing/>
19. Google Cloud Platform Blog: Containers, VMs, Kubernetes and VMware. <https://cloudplatform.googleblog.com/2014/08/containers-vms-kubernetes-and-vmware.html>
20. Google Cloud Platform Blog: Enter the Andromeda zone—Google Cloud Platform’s latest networking stack. <https://cloudplatform.googleblog.com/2014/04/enter-andromeda-zone-google-cloud-platforms-latest-networking-stack.html>
21. Łaskawiec, S.: Proposal: Public IPs per Pod (2017). <https://github.com/kubernetes/community/pull/446>
22. W.M. Kryszicki Włodzimierz, Bartos Jerzy, Dyczka Wacław, Królikowska Krystyna, *Rachunek prawdopodobieństwa i statystyka matematyczna w zadaniach cz. 2.* (PWN, 1999)

Sebastian Łaskawiec has been working as a Senior Software Engineer at Red Hat, focusing on highly scalable, cloud-enabled applications. His main area of expertise are Data Grids, Linux Containers and Network Communication Protocols. In 2016 Sebastian started Ph.D. studies in telecommunications at University of Science and Technology (UTP) in Bydgoszcz.

TACTICS) and he coordinated FP7 project CAMINO on cyber crime and cyber terrorism. He is an author of over 170 reviewed scientific publications.

Rafał Kozik is an assistant professor in the Department of Telecommunication of University of Technology and Life Sciences in Bydgoszcz (UTP). In 2013 he received his Ph.D. in telecommunications from University of Science and Technology (UTP) in Bydgoszcz. Since 2009 he has been involved in number of international and national research projects related to cyber security, critical infrastructures protection, software quality, and data privacy (e.g. FP7 INTERSECTION, FP7 INSPIRE, FP7 CAMINO, FP7 CIPRNet, SOPAS, SECOR, H2020 Q-Rapids). He is an author of over 70 reviewed scientific publications.