Highly Scalable Microservice-based Enterprise Architecture for Smart Ecosystems in Hybrid Cloud Environments

Daniel Müssig[1], Robert Stricker[2], Jörg Lässig[1,2] and Jens Heider[1]
[1]University of Applied Sciences Zittau/ Görlitz, Brückenstrasse 1, Görlitz, Germany
[2]Institutsteil Angewandte Systemtechnik AST, Fraunhofer-Institut für Optronik, Systemtechnik und Bildauswertung IOSB, Am Vogelherd 50, Ilmenau, Germany

Keywords: Cloud, IT-Infrastructure, Scaling, Microservices, Application Container, Security, Authorization Pattern.

Abstract: Conventional scaling strategies based on general metrics such as technical RAM or CPU measures are not aligned with the business and hence often lack precision flexibility. First, the paper argues that custom metrics for scaling, load balancing and load prediction result in better business-alignment of the scaling behavior as well as cost reduction. Furthermore, due to scaling requirements of structural –non-business– services, existing authorization patterns such as API-gateways result in inefficient scaling behavior. By introducing a new pattern for authorization processes, the scalability can be optimized. In sum, the changes result in improvements of not only scalability but also availability, robustness and improved security characteristics of the infrastructure. Beyond this, resource optimization and hence cost reduction can be achieved.

1 INTRODUCTION

The number of enterprises using the cloud to host their applications increases every year (Experton, 2016). Adopting the cloud has in essence two reasons: first, more flexibility and in general better performance. Second, the pay-per-use model is more cost effective than hosting own servers. However, these advantages are only achieved, if the cloud is used efficiently. Thus, the fast provisioning and release of resources should be optimised for each application. Therefore, an application has to be able to scale with the demand.

To specify scalability, the AKF scale cube defines three different dimensions of scaling (Abbott and Fisher, 2015): The *horizontal duplication* produces several clones, which have to be load balanced. The *Split by function* approach is equivalent to splitting a monolith application in several microservices (Newman, 2015), scaling them individually. The last dimension is called *split by customer or region* and applied for optimizing worldwide usage and service-level specific performance. While all three dimensions are compatible, only the first two of them are relevant for our considerations here. Current scaling solutions are using general metrics, namely CPU and RAM, for the *horizontal duplication* and load balancing. A general drawback of these approaches is that they are not able to describe the utilization of a service precisely. E. g. services with a queue have mostly a stable CPU utilization, no matter how many requests are in the queue. There are already metrics using the queue length, but therefore each entity would need to have the same run time.

Another issue with efficient scaling of cloud infrastructures is connected to the high priority of security aspects and the rigorous implementation of security and authorization patterns. Looking at conventional authorization patterns, which are quickly reviewed in the paper, their scaling behaviour turns out to be limited due to the need of replication of services which are only used for the authorization process. It further turns out that there is a lack of several security objectives such as availability or robustness.

We introduce and describe an infrastructure, which is able to dynamically scale itself according to the used resources and predicted resource consumption. It is not only able to start and stop replicas of services, but also compute nodes. This is achieved by using custom metrics instead of general metrics.

To address the inherent scaling problems of conventional authorization pattern, we propose a new design approach for authorization in microservice architectures, which is able to utilize the benefits produced

Müssig, D., Stricker, R., Lässig, J. and Heider, J.
Highly Scalable Microservice-based Enterprise Architecture for Smart Ecosystems in Hybrid Cloud Environments.
DOI: 10.5220/0006373304540459
In *Proceedings of the 19th International Conference on Enterprise Information Systems (ICEIS 2017) - Volume 3*, pages 454-459
ISBN: 978-989-758-249-3

by the scaling infrastructure by avoiding scaling of –non-business– services. Our distribution concept even results in better partition tolerance and has the ability to achieve lower response times. In sum, the approach should lead to a more efficient scaling of the cloud infrastructure, which is applicable in almost any cloud environment.

The paper is structured as follows. In Section 2 we refer to current related approaches. Section 3 describes the key concept and essential services for a flexible infrastructure design. Afterwards we describe custom metrics that are used for optimized scaling of the application in Section 3.3 and in Section 3.4 we illustrate how highly scalable infrastructures can be optimized by using machine learning algorithms. In Section 4 we show weaknesses of common authorization patterns and introduce details of a new approach. The paper concludes with Section 5.

2 RELATED WORK

To the best of our knowledge, this is the first attempt of a highly scalable enterprise architecture for the cloud which is optimized with security by design instead of inhibition through security restrictions. Toffetti et al. describe an architecture for microservices, which is self-managed (Toffetti et al., 2015). This architecture focuses on health management as well as auto scaling services. However, this work is based on etcd[1] and does not describe a generic scalable microservices architecture. Furthermore, this paper disregards the scaling of compute nodes and security aspects. The API key distribution concept seems to be similar to the whitelist configuration apparently used by Google for the Inter-Service Access Management[2]. A major difference is the ability to update permissions during run time by using an observer pattern in our approach. Additional security aspects like network traffic monitoring and intrusion detection can also be done in a microservice architecture (Sun et al., 2015). They use the perspective on a microservice environment as a type of a network as we do. Using docker container for our approach we refer to (Manu et al., 2016) for docker specific security aspects. The potential scope of application is quite extensive. E. g. (Heider and Lässig, 2017) describe a development towards convergent infrastructures for municipalities as connecting platform for different applications. The authors outline the need for high scalability and security as there are requirements for high computational power and extensive data exchange in many use cases, if different platforms are tied closely together and considered as connected infrastructure landscape.

[1]https://github.com/coreos/etcd

[2]https://cloud.google.com/security/security-design/#inter-service_access_management

3 EFFECTIVE INFRASTRUCTURE-MANAGEMENT

Running an infrastructure is connected with high costs and administrative effort. As mentioned before, the number of enterprises migrating to the cloud is increasing, since the cloud promises advantages in cost and administration efficiency. In this section we present our approach of an intelligent infrastructure, which is capable of up- and down-scaling the number of used compute nodes as well as service instances. The model is very general and can be deployed analogously in many use cases.

3.1 Services

The proposed architecture consists of several services, which are operational. These are shown in Figure 1. The services run similar to the bussiness

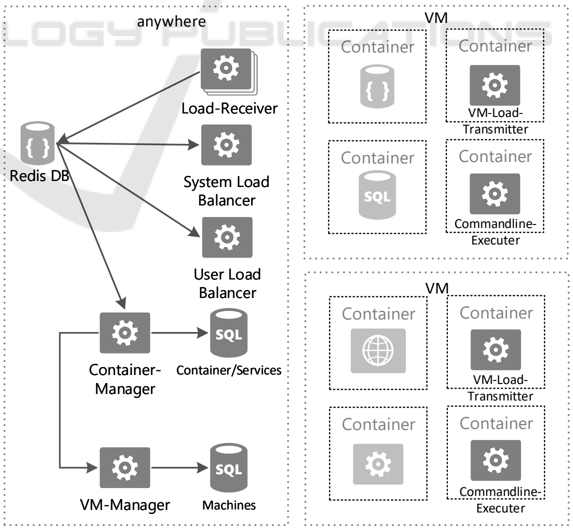

Figure 1: The architecture is managed by six operational services. Each service sends its utilization to the load-receiver, which stores the information in an in-memory db. The loadbalancer uses the information to balance the requests and the container manager is scaling the services. To scale the number of compute nodes, the information of the VM load transmitter is used. The commandline executor receives instructions from the machine manager to create or delete compute nodes.

services in their own container and are divided in the groups: node-specific services, self-contained services and system services. While the services of the first group, such as the load balancer and the databases, run on specific compute nodes, services of the second group are independent from the node they run on. The load-receiver, container-manager and machine-manager belong to this group and can run on every compute node. The third group contains services which are running in addition to the application services on each compute node. These services are the VM-load-transmitter and the commandline-executor. We assume, that there is an interface, which could be used to create and delete compute nodes. Most cloud providers and private cloud frameworks such as OpenStack already offer similar services.

The commandline-executor is used to create and delete containers. This service receives commands e. g. via a POST-REST call and executes them on the compute node. Only this service can access the container engine of the compute node. To ensure that there is no unauthorized usage of the service, it is only accessible by the container-manager.

The other service which has to run on every compute node is the VM-load-transmitter. This service monitors the utilization of the compute node. The measurements of cpu, ram, bandwidth and used disk space are sent to the machine-manager. These information are used to determine if machines have to be started or stopped.

The machine-manager stores information about the environment. In most cases there are many different types of compute nodes which can be ordered or created. They differ in the amount of processors, their performance, ram, disk space and disk type. The available types have to be defined and configured. The service stores the commands to start or stop/delete such a compute node. Besides this the service stores also the maximum number of compute nodes of the same type that are allowed to run. This mechanism is intended to restrict the costs. It is also possible to store cost limits, e. g. the maximum amount to be spent per month.

Once the machine-manager starts a compute node, the service stores information about it. These consist of its IP address, but also the type of compute node and whether the compute node is allowed to be shutdown or not. This design feature is important since the infrastructure requires some static parts, e. g. machines to run databases or to store certain information in general. Besides indicating the application of a machine, its type is also used to determine the compute node, on which an instance of a service should be executed. E. g. databases usually require fast disks to run with good performance, while a machine that is running compute services needs a cpu with more performance. The machine-manager stores statistics about the compute node e. g. cpu utilization, used ram, bandwidth and used disk space. All information should be stored with time stamps, as we explain in detail later.

The container-manager controls the services. As initial step, every new service has to be registered. This can be part of the deployment process. For a registered service at least the following information should be stored: name, minimum number of instances, maximum number of instances, the start and stop commands, bool value if load balancing is required and requirements for the node (e. g. the name of a particular compute note, if specified). For an efficient management of the instances, also information about the resource consumption should be stored. If an instance of a service is executed, the IP of the machine and the port where the service can be reached, should be stored in a service registry. The IP address also corresponds the commandline-executor. If the average load of all instances of a service is above or below a certain percentage in a given time frame, the container-manger can take action and scales the service up or down. The container-manager sends a request to the machine-manager to decide on which compute node the instance should be started or stopped. In general the container-manager stores information about the utilization of containers using custom metrics and the number of instances of a service, which is discussed in detail in Section 3.3.

3.2 Load Balancing

The load balancer and the load-receiver are using the same database, preferably an in-memory database such as redis[3]. The latter receives the utilization of the custom metric from the instance and stores it in the database. Redis supports sorted sets. This makes it possible to store the information in the right order for the load balancer, so that reading from the database has the least effort. We are using a set for each service, resulting in multiple scoreboards.

For load balancing a NAT proxy with feedback method can be used. With this method the load balancer can handle requests with different durations much better than other approaches such as round robin. But also due to the frequently starting and stopping of machines and containers other methods apart from this and the round robin approach wouldn't work. The load balancer and the in-memory database which stores the utilization should run on the same

[3]https://redis.io

compute node to lower the latency. With increasing number of requests the load balancer has to be scaled. There are already approaches such as (Shabtey, 2010). However we consider to use a different approach. We want to use, if needed, a load balancer for user requests and a load balancer for inter service communication. If it has to be scaled further we recommend to use a load balancer for each heavy used service to avoid having more than one load balancer per service. Since the requests are referred in our architecture to different instances of services a special security pattern is needed, which authenticates each request. A further discussion on this topic is given in Section 4.

3.3 Custom Metrics

Recently, there are many articles which describe that only a small number of services or applications could be efficiently scaled using a CPU and/or RAM metric. To encounter this, our approach uses custom metrics, which are defined for each service separately. This is done by the developers of the service, since they have most knowledge about it. So they can define it e. g. as the internal queue length or progress of an algorithm. The service sends than a percentage between 0 and 100 to the load-receiver in a self defined interval. Utilization of services should be between 60 and 80 percent. When there is only one replica it is scaled up at 60 percent. The percentage where a scaling is done increases with the number of replicas. The steps could be also defined in the container-manager. Since it takes some time to rebalance when an instance is started or stopped, the developers also define a warm up and cool down time for the service. The former describes the time which is needed for starting a new instance and distribute new incoming requests equally. To the contrary, the latter describes the time which is needed on average to stop an instance after all pending requests on this instance were finished.

3.4 Enhanced Infrastructure Management

Infrastructure management could be enhanced further using machine learning techniques (Ullrich and Lässig, 2013). Since the services collect information about the number of instances and machines as well as their utilization with time and weekday, maching learning algorithms can learn pattern and operate in advance. This could be particularly useful for online shops, but could be carried over to the most other web-based services.

Besides this, machine learning algorithms could also be used in other directions. Very important when scaling services is the information about the duration till the new instance is started and completely integrated in the balancing. This is different from service to service and depends even on the compute node and where it is started. Furthermore, the machine-manager learns the usual utilization of CPU and RAM to enhance the decision on which a new instance should be started to use the resources optimal.

4 SECURITY CHALLENGES

A flexible infrastructure architecture as presented in this paper realizes a dynamic environment, which practically ends up in a complex system of service instances that can be removed or newly created immediately and which are sending or receiving requests permanently. One of the security objectives (in addition to the CIA triad) that have to be considered is authorization, which is necessary for protection against misuse. Accounting for the distributed character of the services, the CAP-Theorem has to be considered as well. We investigate some common authorization principles concerning the compatibility with our presented infrastructure and point out their weaknesses concerning other security objectives and system abilities. Afterwards we propose our idea which is optimized to maximize the robustness, availability and the scalability of each microservice.

4.1 Common Authorization Principles

The task of authorization is to answer the question if a request to a microservice has to be fulfilled or rejected. There are several possibilities to design the authentication and authorization process such as shown in Figure 2.

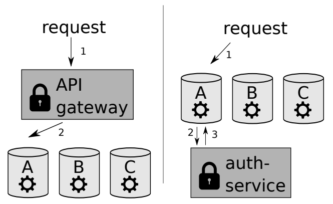

Figure 2: Common design patterns for authorization in a microservice architecture with weaknesses in availability, scalability and robustness.

In the first part of Figure 2 the API gateway pattern[4] is shown. In security contexts the gateway is called Application-Level-Gateway (ALG). All requests were sent to the API gateway which routes

[4]http://microservices.io/patterns/apigateway.html

them, creates access tokens, encrypts messages, etc. There are significant advantages of this approach. The real interface addresses (URLs) of the microservices can be hidden, injection inspection or input validation (content-types, HTTP methods) can be realized equivalently for each service. But for the authorization process this service requires also a database containing data for all services, which creates vulnerabilities.

Also the second design approach is often used for microservices. The request is sent directly to the service which is able to fulfill it. The service itself sends a new request to an authorization service (*auth-service*), which checks whether the user (or other service) has the permission to use the microservice (A). If the requester is privileged, the microservice calculates the response. One advantage of this process chain is the separation of sensitive user data from the open interfaces, another advantage is the less extensive functionality of the *auth-service* compared to an API gateway, which improves the authorization process concerning response time. Tasks from the gateway such as validation has to be addressed by the microservices additionally. A negative aspect is the generation of traffic from the service (A) to the *auth-service* for each incoming request. If an opponent sends many requests during a (D)Dos attack, the pattern supports him by multiplying each request. Moreover, an attack on service A even attacks the *auth service* and can make it unavailable for the other services. These and similar designs contain one single point of failure and are vulnerable to DoS-Attacks on one single service. The whole application is affected if the *management service* is not available. Accounting for the dependency of the microservices from the *management services*, they have to be scaled together. The worst aspect of these designs is the dependency between the microservices and the management services from the count of requests, not of running service instances.

4.2 API-Key-Distribution

Following our requirements, each microservice should be able to fulfill a request without connecting to another service. This makes the application not only more secure but also faster. Hence, each service requires a database to store information about valid requesters. In more detail, each instance of a service requires this data which consists of an API key and maybe some additional information, such as the role connected with a key. The key represents a service or user, which is authorized to use it. We divide the lifetime of a service instance into two parts, the initialization and the production phase. The initialization phase starts immediately after the creation of a new instance and the *API Key distribution* has to be done. This is shown in the left part of Figure 3.

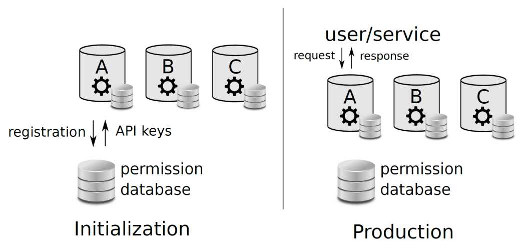

Figure 3: The two phases of API key distribution to realize independence of each microservice from additional *management services* while fulfilling a request.

The newly created service instance registers on the permission database and receives the API keys, there is merely read-only access required. Note that several instances of the same service get the same data.

In the production phase as shown in Figure 3, the microservice is available for requests, which contain the authentication data (API key) of the user or service where the request comes from. Now our prepared service instance is able to authorize this request without connecting to other *management services* and the response can be calculated immediately.

The services can easily be scaled by creating new instances and going through the initialization process. Maybe there is also a *management service* required if direct communication with the database has to be avoided, but this service is only essential during the initialization phase. Afterwards there are no consequences for the running instances if the service is not available. Another difference is that the application services only has to be scaled if the number of requests is rising. The access to the permission database depends only on the number of instances, not the number of incoming requests. During the production phase, changes in permission settings (adding a new user for example) will be sent from the permission database to the service instances based on the registering process during the initialization phase, which is an observer pattern. At the end of the production phase (removing the instance), the service instance must be unregistered from the permission database. Advantages of this approach besides improved scalability and availability are reduced network traffic for incoming requests and improved robustness.

The costs for these benefits are situated in terms of consistency, because changes in permissions are becoming active with a delay. With focus on requests from each service to another, the availability has higher priority than consistency, because permissions did not change permanently.

4.3 Secure Data Transmission

The network traffic (user to service and service to service communication) can possibly be sniffed, changed and interrupted by a man in the middle. The assumption of an existing active adversary leads to serious problems in secure communication, because we can not exchange encryption keys between newly created instances. The Diffie-Hellman key exchange is vulnerable to this kind of attack as well (Johnston and Gemmell, 2002), due to the lack of authentication.

Required is an *information advantage*, which must be pre-distributed over a trusted channel (Goldwasser and Bellare, 2008). In the three party model there is an *authentication server* which shares private keys with each party and generates session keys for each communication session. This has the disadvantages of a centralized service. If we assume that every service instance receives its own API key over an trusted channel (e. g. in the Docker image) and the permission database contains only non-compromised data, the API keys can be used as information advantage.

It is not necessary to authenticate each instance, because the following authorization process is based on the service level as described in Section 4.2. The API key could also be used as secret for signing with a HMAC, which can be used to authenticate a service and verify the integrity of a request. Using this approach we are able to use key exchange methods, which results in the ability to ensure the confidentially of a request.

5 CONCLUSION

In this position paper we describe the concept of a high scalable microservice infrastructure using custom metrics in addition to common CPU and RAM measurements. It uses resources more efficiently for reducing costs in the public cloud and fewer workload in the private cloud. The different operational services necessary for our approach can be expanded with smart machine learning algorithms for self-optimization and self healing.

The paper also proposes an authorization pattern for the proposed microservice architecture, which supports not only the scalability of our flexible infrastructure but also security objectives such as availability and robustness.

We plan to implement our suggestions in a framework which can be used easily to implement and optimize a microservice architecture. Afterwards we plan to apply the framework for an evaluation based on different use cases and prototype implementations in the Internet of Things context, such as Industry 4.0 or Smart Home applications. The performance and flexibility of the approach must be evaluated and compared to other approaches based on different benchmarks.

REFERENCES

Abbott, M. L. and Fisher, M. T. (2015). *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*. Addison-Wesley Professional, 2nd edition.

Experton (2016). Marktvolumen von Cloud Computing (B2B) in Deutschland nach Segment von 2011 bis 2015 und Prognose für 2016 (in Millionen Euro).

Goldwasser, S. and Bellare, M. (2008). Lecture notes on cryptography. *Summer course Cryptography and computer security at MIT*, 1999:1999.

Heider, J. and Lässig, J. (2017). Convergent infrastructures for municipalities as connecting platform for climate applications. In *Advances and New Trends in Environmental Informatics*, pages 311–320. Springer.

Johnston, A. M. and Gemmell, P. S. (2002). Authenticated key exchange provably secure against the man-in-the-middle attack. *Journal of cryptology*, 15(2):139–148.

Manu, A., Patel, J. K., Akhtar, S., Agrawal, V., and Murthy, K. B. S. (2016). Docker container security via heuristics-based multilateral security-conceptual and pragmatic study. In *Circuit, Power and Computing Technologies (ICCPCT), 2016 International Conference on*, pages 1–14. IEEE.

Newman, S. (2015). *Microservices: Konzeption und Design*. mitp.

Shabtey, L. (2010). US Patent No. 7,739,398 B1: Dynamic Load Balancer.

Sun, Y., Nanda, S., and Jaeger, T. (2015). Security-as-a-service for microservices-based cloud applications. In *Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on*, pages 50–57. IEEE.

Toffetti, G., Brunner, S., Blöchlinger, M., Dudouet, F., and Edmonds, A. (2015). An architecture for self-managing microservices. In *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*, pages 19–24. ACM.

Ullrich, M. and Lässig, J. (2013). Current challenges and approaches for resource demand estimation in the cloud. In *IEEE International Conference on Cloud Computing and Big Data (IEEE CloudCom-Asia 2013)*, Fuzhou, China.