Received: 27 December 2018 | Revised: 18 May 2020 | Accepted: 6 June 2020

DOI: 10.1002/spe.2877

RESEARCH ARTICLE

WILEY

A framework for microservices synchronization

Antonio De Iasio | **Eugenio Zimeo**

Department of Engineering, University of Sannio, Benevento, Italy

Correspondence
Eugenio Zimeo, Via Traiano 1, 82100 Benevento, Italy.
Email: zimeo@unisannio.it

Funding information
GAUSS PRIN 2015, Grant/Award Number: 2015KWREMX; SISMA PRIN 2017, Grant/Award Number: 201752ENYB_002

Summary

Microservices architecture and continuous software engineering are becoming popular approaches for developing and operating software products. The enabling feature of this success is the independence of the execution environments hosting microservices: by insulating failures and versioning in specific microservices, a complex application benefits of high availability at runtime and agility at development time. However, execution independence does not mean functional independence. Microservices need to interact among them to fulfill a common goal of an application. The unavailability of a microservice may seriously impact other dependent microservices, limiting continuity. To address this new kind of problem in microservices architecture, we argue the necessity of a synchronization mechanism able to support microservices coordination according to their running states: dependent microservices should wait for unready ones to avoid useless and faulty interactions. In this article, we propose a new framework, *Synchronizer*, able to support synchronization among microservices by exploiting distributed registries for collecting health/state information about deployed containers and hosted microservices. It has been implemented for the OpenShift platform and validated in different use cases: for example, for coordinating applications bootstrap and for programming scripts of continuous deployment orchestrators, such as Jenkins. In both cases, *Synchronizer* worked as expected and showed the positive effects of synchronization, giving us a valuable feedback about the possibility of further extending its application and of integrating the feature in existing microservices frameworks (eg, services mesh).

KEYWORDS

concurrency, continuous deployment, microservices paradigm, middleware, services coordination, synchronization

1 | INTRODUCTION

Microservices paradigm aims at designing applications as sets of small loosely coupled services, which adhere to the bounded-context pattern.[1] Each service runs in its own process and communicates with each other by means of well-defined interfaces and lightweight mechanisms, such APIs (Application Programming Interfaces) based on HTTP.[2] Microservices carry along many benefits including technology heterogeneity, resilience, scaling, and ease of deployment.[3] Moreover, unlike monoliths, microservices applications can be managed at a finer grain by replicating only the overloaded services.[3,4]

The constituent microservices of an application need to interact and this may not occur safely. In some situations, as when a microservice functionally relies on other ones to carry out its task successfully, communication may need to happen under some synchronization constraints. In fact, despite its execution and deployment independence, a microservice may not always be available or even if so, it may not yet be ready to perform its task. This could lead to unsafe interactions and consequent errors in different phases of a microservice life-cycle. The importance of this problem is proved by pre-existent solutions, such as the circuit breaker pattern. This pattern prevents faulty interactions by opening the circuit, that is, cutting off the connection between dependent microservices when a request fails, and by closing the circuit, that is, reallowing the connection when, after a timeout, the request is retried successfully.[5]

Problems in coordinating microservices may arise at different levels. During the application bootstrap, for example, the start-up order should follow the dependencies among the constituent microservices in order to avoid faulty interactions; in DevOps, a meta-application based on microservices, typically used to automate the building, and deployment phases of software life-cycle, could suffer of the same coordination problems affecting a conventional microservices application when different services need to be coordinated (eg, a microservice under test and the testing microservice).

This article proposes a solution to the problem of coordination among microservices based on a synchronization framework enabled by means of a platform service, named *Synchronizer*. *Synchronizer* has been designed to address different kinds of synchronization problems and to support functional and performance tests in DevOps pipelines. Coordination and synchronization among microservices is a general problem that could affect applications and infrastructures in different scenarios; however, in this article, the focus is mainly on applications bootstrap and on DevOps continuous deployment pipelines. The analysis conducted in the two scenarios allowed us for (a) validating the framework effectiveness in removing synchronization violations and (b) studying interesting related problems whose solutions we will take into account for a future extension of the framework.

The article advances the state-of-the-art by proposing a theoretical and practical solution to the problem of synchronization violations in applications characterized by large sets of components with a high degree of execution autonomy as the ones based on microservices.

The rest of the article is organized as follows. Section 2 introduces some motivating scenarios to highlight the importance of the problem addressed in the article. Section 3 introduces background concepts. Section 4 presents the problem statement and introduces a conceptual definition of the proposed framework. Section 5 presents the implementation of *Synchronizer* and reports on the results of tests aiming at assessing the validity, usefulness, and correctness of the proposed mechanism. Section 6 positions the contribution with reference to related works. Finally, Section 7 concludes the article and highlights future research directions.

2 | MOTIVATING SCENARIOS

Usually, microservices of an application are deployed independently, but typically they still functionally rely on other ones. During the bootstrap, for example, a careful planning of such phase would be necessary to avoid turning the benefit of execution independence into a potential hazard. To this regard, the authors in this article[6] suggest tracking dependencies by using a Directed Acyclic Graph, yet their approach does not provide any robust solution preventing dependencies problems from disrupting the bootstrap.

To understand the problem, consider an application (see *Sock Shop*[7] in Figure 1) that needs to be bootstrapped: constituent microservices start-up order should be constrained as to avoid early interactions which could bring about faults. For example, *catalogue* may contact *catalogue-db* when this is not ready yet, which would result in an error, also likely to propagate to other microservices. At the same time, after the bootstrap phase has successfully completed, the same interactions should still be somehow coordinated so as to avoid problems when some microservice, another service relies on, goes down for some bug or maintenance activity.

Moreover, as any software architecture paradigm, even microservices have their own set of practices.[8] DevOps is such an example that integrates *development* and *operations*, as in *Continuous Integration* (CI),[9] *Continuous Delivery* (CDE),[10] and *Continuous Deployment* (CD).[10,11] They are hierarchically connected: CD implies CDE, which in turn implies CI. In particular, CD means that the whole building and deploying pipeline, unless any test fails, is automated up to the release to the customer. This automation can be easily supported by microservices that turn CD into a sort of *meta-application*. Therefore, CD pipelines[12] can either be implemented through a traditional monolith application or could be themselves meta-microservices applications. In this case, as shown in Figure 2, each phase could be coded in a different

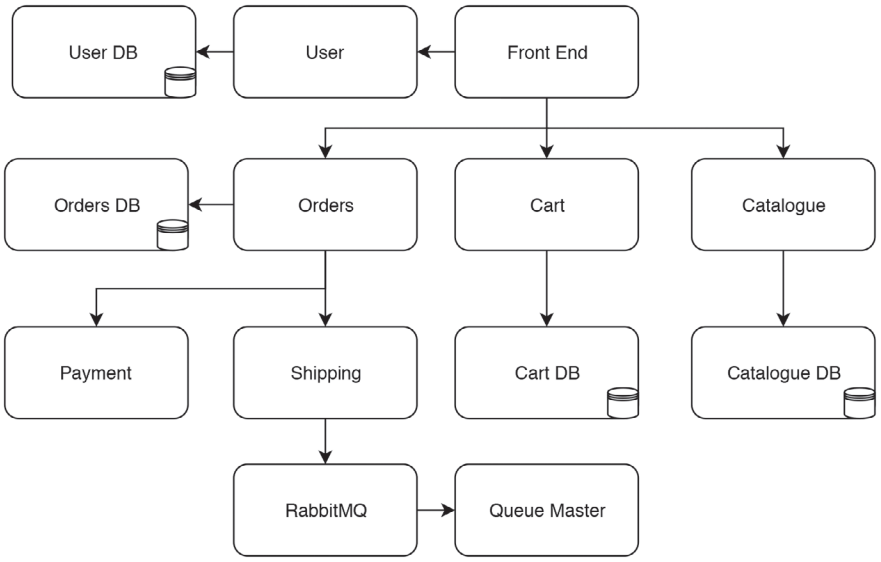

FIGURE 1 Example of a microservices application

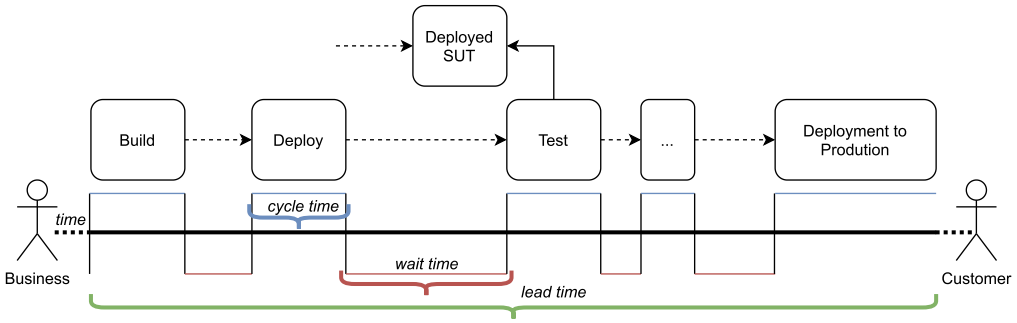

FIGURE 2 Lead time in a CD pipeline [Colour figure can be viewed at wileyonlinelibrary.com]

microservice and it would become important to synchronize the transitions between consecutive phases, that is, between each corresponding microservice.

Moreover, as the output of a phase could be an application microservice, it would become also necessary to synchronize the latter with the subsequent phases. For example, as shown in Figure 2, a possible failure scenario would involve starting up the microservice in charge of testing only when the deployed Service Under Test (SUT) becomes ready. In fact, in this case, a lack of synchronization could lead to the failure of the whole pipeline, which should be executed again from a checkpoint or, in the worst case, right from the beginning. Even more, in case a performance test must be executed, the testing microservice should not be run unless all replicas of the deployed SUT are ready, lest test validity may be affected.

Therefore, in spite of its high potential automation level, CD based on microservices technologies introduces potential flaws: CD pipeline's stages may be run concurrently and complex orchestration patterns may be identified and used. For these reasons, the need to carefully plan how each stage communicates with its neighbors becomes more evident and a synchronization mechanism becomes necessary. However, to carefully evaluate how such mechanisms help improve and keep the pipeline healthy, appropriate metrics must be identified.

Several metrics[13-15] gauge single phases of a pipeline, but *lead time*, defined as the time from one feature planning until its use in production,[12,16-19] deals with the whole pipeline. For this reason, it is a first class metric that allows to estimate the overall efficiency of a pipeline. Moreover, as shown in Figure 2, lead time can be further decomposed in cycle times and wait times.[12,16-18] Cycle time is the time spent in a single stage, whereas wait time is the dead time in between two stages. As synchronization violations may badly affect the lead time by increasing its wait time component, this metric appears to be useful to analyze the problems with a continuous deployment pipeline and to validate the solution.

Therefore, to solve the synchronization problems inherent to application bootstrapping and to enhance the implementation of CD pipelines, we argue the necessity of a synchronization mechanism to deal with coordination issues.

3 | BACKGROUND

This section briefly introduces some microservices-enabling technologies (mainly containers and pods) with the aim of better clarifying their execution independence and the impact on applications execution. The terms and the concepts introduced are also important to better understand the problem statement and the related discussion.

Containers are virtualization technologies that, unlike virtual machines, work at the OS level.[20] While virtual machines handle their own operating system executed on top of a virtualization layer managed by an hypervisor, containerization uses Linux kernel primitives to isolate processes and their resources in the context of the same OS. The lack of an hypervisor for the containerization is the main difference. Containers host single applications and their related libraries and drivers while share OS kernel and other resources. Therefore, they take less space and run faster: while VMs (Virtual Machine) take seconds to boot, containers take milliseconds. Such difference is fundamental, as DevOps-oriented environments require short boot times. Moreover, self-containment, easy replacement, and evolution make containers very recommended in highly dynamic contexts, such as applications in cloud governed by DevOps practices.

A container alone is not enough to build a microservice. Since each container should execute only one specific task, microservices that need to carry out more complex tasks may need to be backed by two or more containers. Also, to improve performance, a microservice should provide its constituent containers with shared storage capabilities and local communication. In that case, the constituent containers can be treated as a single cohesive unit as concerning non-functional requirements (scalability, reliability, and so on). Kubernetes,[21] for example, implements this kind of aggregation in the concept of Pod.

A Pod groups containers together and provides them with storage and network sharing. In general, they serve as unit for deployment, horizontal scaling, or replication: therefore, even a single container can be wrapped in a Pod. Colocation (co-scheduling), shared fate (eg, termination), and dependency management are handled automatically for containers in a Pod.[21] Some containers may serve specific purposes; for example, by using the sidecar design pattern, *init containers* are run before application containers and may be used to perform preliminary operations or to delay the startup of application containers until some preconditions have been met.[22]

Pods are mortal and once terminated, they have to be created anew and receive new network addresses and IDs. However, logical microservices should survive to their constituent Pods termination to ensure other services can keep track of them; for this reason, a further higher-level abstraction is provided: *Service*, which can group together a logical set of Pods, as shown in Figure 3. Therefore, microservices assume their properties, such as their network addresses, regardless of their constituent Pods. For example, Pods can be replicated and dynamically added to the service according to the external load or to other parameters. However, microservices are not statically bound to network addresses but are decoupled from them through a naming service (implemented by a registry) allowing their dynamic discovery.

Without lack of generality, we can say that each microservice may be composed by a *Service/Endpoint* (that Kubernetes calls *Service*) and by a variable number of Pods. *Service/Endpoint* provides smart endpoint management and transparent scalability by hooking multiple instances of the same Pod; Pod, instead, provides the execution environment, that is, the core of the microservice. However, in case scaling properties are not needed, the very microservice can be identified by its single constituent Pod; as a consequence, in this situation, an application Pod matches a domain microservice and a meta-Pod matches a system microservice.

In any case, even though the aforementioned features enable high flexibility, they may still become source of several issues in fast-paced environments. In fact, as Pods can be created and destroyed very easily and run in a high number of instances, the way their parent microservices communicate should be carefully engineered as to avoid situations where a microservice may need to contact another one, which is either not available yet or backed by an insufficient number of Pods.

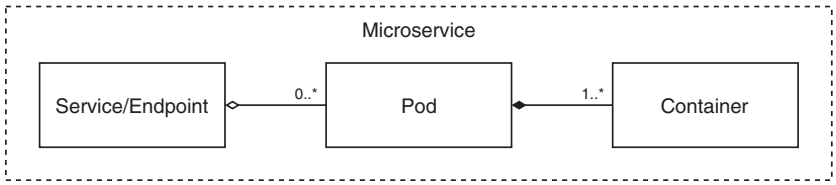

FIGURE 3 Relationship among services, pods, and containers

4 | MICROSERVICES SYNCHRONIZATION

In this section, we present the problem of microservices synchronization and devise a general framework to deal with it.

4.1 | Problem statement

Despite their high autonomy, microservices can be spatially coupled for satisfying functional dependencies; this, joined to concurrent development and deployment, could lead to synchronization faults. In different situations, either choreography or orchestration[22,23] could mitigate such issue; yet, synchronization problems may still appear and disrupt the correct operation flows both between microservices of the same class and between microservices of the domain and system classes. For example, as shown in Section 2, domain microservices should be synchronized during the bootstrap phase; whereas in a continuous deployment pipeline, system and domain microservices should instead be synchronized.

Both scenarios could be implemented through choreography or orchestration; as a matter of fact a need for synchronization may arise in either of these two mechanisms. For this reason, we strive to provide a framework which would fit them both. However, a decentralized process, such as microservices application bootstrapping, is usually dealt with choreography as each microservice is launched independently; on the other hand, a centralized process, such as a CD pipeline, is generally implemented through orchestration as the latter enforces a global view of the synchronization itself. Therefore, for the sake of completeness, in the first scenario the synchronization is dealt with choreography, whereas in the second with orchestration. In either case operational failures happen when a microservice is not backed by the expected number of Pods. In fact, Pods have their own life-cycles, which means they go through a series of phases, depending on a set of conditions. In particular, once deployed, they may enter the running phase but can be either Ready or Not Ready to perform their operations.

Figure 4 helps us to generalize microservices interactions in terms of synchronization problems. In particular, it shows two microservices, which, without lack of generality, can safely be identified by their single constituent Pods, *Pod A* and *Pod B*. Both are up and running and can either be in a Ready or Not Ready state. The Not Ready state can be either the consequence of some temporary failure or the need for carrying out preliminary actions, as when some configurations should be loaded from a database.

The numbered circles help define a logical sequence of actions overlapped with the temporal line. In (a), at *1*, Pod B should wait for Pod A to become ready in order to avoid failures. This happens because Pod A's preliminary operations last longer than Pod B's corresponding ones or because Pod B depends on Pod A. In (b), the synchronization among the

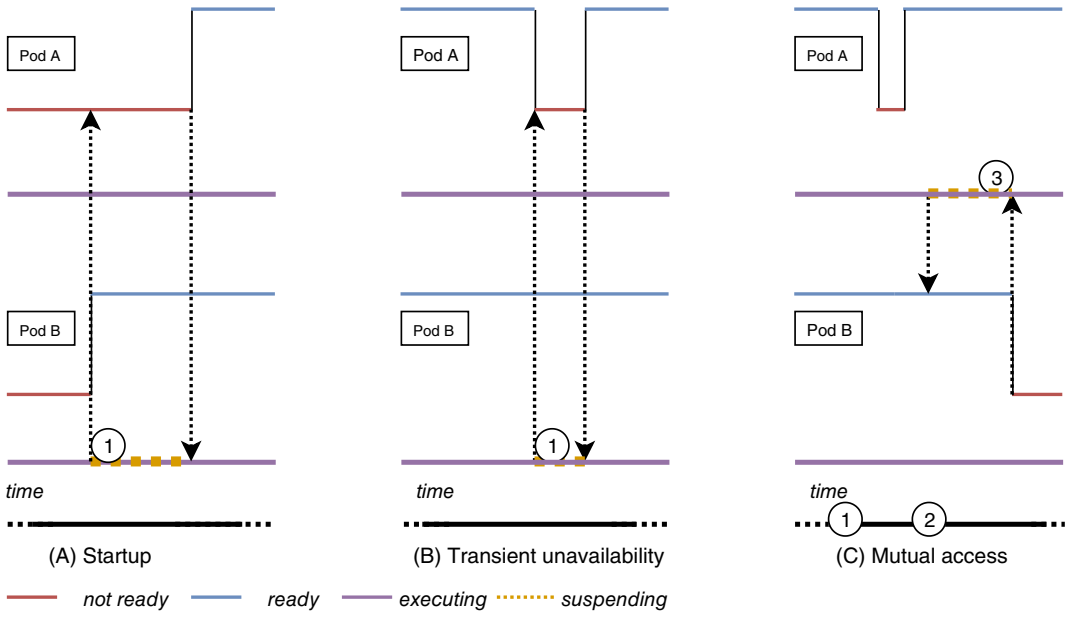

FIGURE 4 Pods synchronization problems [Colour figure can be viewed at wileyonlinelibrary.com]

two Pods must be kept for some time interval, that is a Pod depends on the other Pod to keep fulfilling its task. At *1*, Pod A becomes unready for some kind of failure and Pod B should wait for Pod A becoming Ready again. Finally, in (c) a rather extreme situation is illustrated. From *1* to *2*, both Pods can run independently. At *3*, instead, one Pod at a time can be executing in the Ready state. For this reason, Pod A is suspended until Pod B goes into the Not Ready state. This could happen in a low-memory environment where a memory-consuming Pod may wait for the other one to become Not Ready before performing some intensive tasks.

Different classes of problems for synchronization among Pods can be identified on the basis of the previous analysis:

- *start-up*—a Pod has to wait for another Pod to complete some preliminary boot operations;
- *transient unavailability*—a Pod has to wait since the Pod it communicates with is temporary down;
- *mutual access for governing resources*—a Pod has to wait for another Pod to go into the Not Ready state in order to free shared resources.

Hence, we need a general synchronization framework able to support the aforementioned classes of problems. In this article, we focus on devising a solution to the *start-up* problem class and on testing its validity through two representative problem instances, (a) *application bootstrapping* and (b) *continuous deployment pipeline*. In the first case, synchronization should be decentralized and operate as to suspend the start-up of microservice that depends on other ones until the latter have become ready. It should not work to improve absolute temporal constraints but rather to enforce a relative order which does not break dependencies relationships. Consequently, to assess the correct bootstrap process, we need to know the relative order of start-up of all the involved microservices.

Synchronization can either be seen as an alternative to the circuit breaker pattern or as a solution for its improvement. In fact, considering the example described at the beginning of this section, in Figure 5 microservices start-up order is not constrained and a circuit breaker is used to prevent useless interactions between *catalogue* and *catalogue-db* at the cost of several requests flowing needlessly through the network. In fact, no matter the state of the circuit, requests from the *client* would still be propagated up to the *catalogue* with a consequent network overloading, which could increase even more as the chain of requests becomes longer and the breaker is not appropriately placed. However, after waiting for a preset timeout, the circuit is closed and requests are again forwarded to *catalogue-db*, which, if ready, will now be able to respond. Since a higher threshold of failed requests can be set to trigger the circuit, opening it after the first failed request is a simplification, which we have adopted to keep the figure easily readable; yet, it does not invalidate the fact that a circuit breaker implies a trial-and-error mechanism, which we aim at overcoming.

On the other hand, equipping every dependent microservices with its own synchronization mechanism (see Figure 6) allows their start-up to be delayed according to their dependencies. In particular, in this case, *client* cannot incur in an error, as it will not be started up unless *front-end* is ready, which would mean that also *catalogue* is ready and so on in this hierarchical fashion from the outermost to the innermost dependency. In this way, we avoid handling specific errors when the breaker is open and provide a more robust mechanism to prevent faulty interactions when some services go temporarily down because their start-up order is constrained a priori by their dependency relationships.

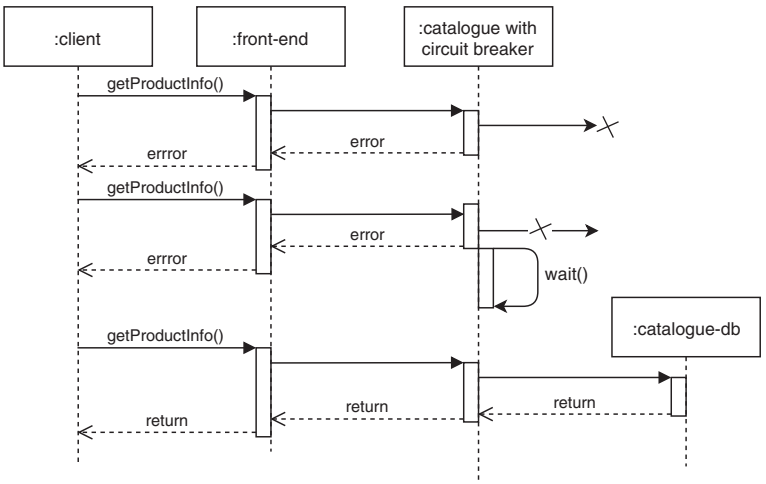

FIGURE 5 Bootstrapping with a circuit breaker

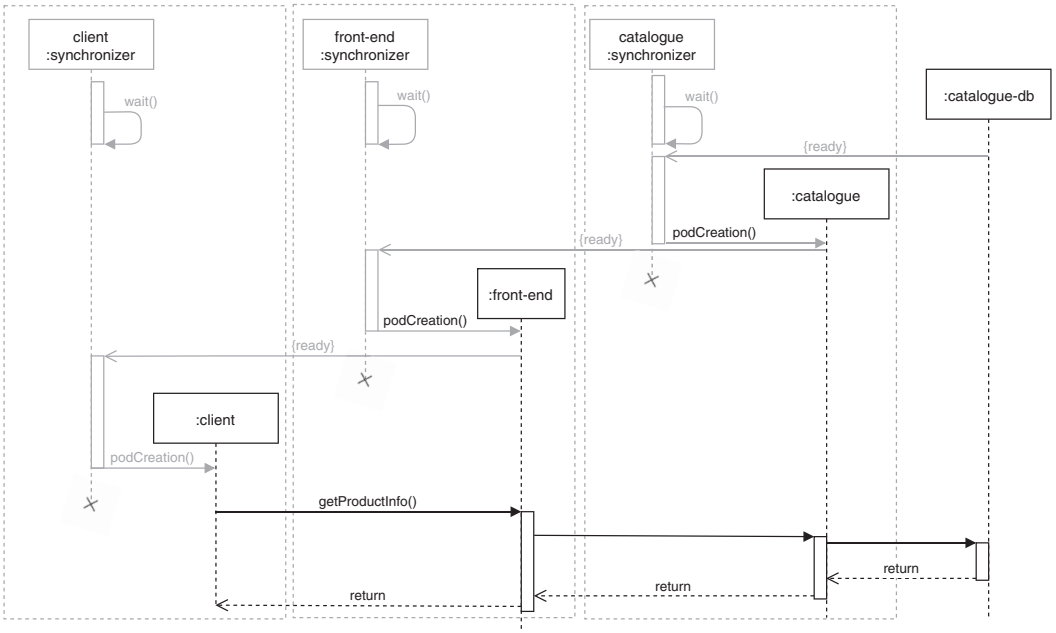

FIGURE 6 Bootstrapping with *Synchronizer*

Synchronization has a beneficial effect also for network traffic reduction. In fact, when a polling mechanism is used, the main parameter to play around with is the retry interval. The tighter this interval the more tries fail but a tighter synchronization is achieved. On the contrary, the wider this interval the fewer tries fail and consequently a looser synchronization is achieved. This implies a compromise between the amount of failed tries, that is the network traffic overload, and the possible synchronization error we are willing to accept; this trade-off should be accurately resolved on the basis of the specific application, further hindering the flexibility of a polling mechanism. Moreover, depending on the complexities and the amount of the services involved, the network traffic overload could increase accordingly; in fact, whenever a service needs other services to carry out its task, the amount of messages needlessly flooding the network should be multiplied by the number of internal services involved. This would also mean that in order to improve the performances of the polling mechanism, the trade-offs should be individually resolved for each service, according to its specific communication footprint.

Circuit breaker can only mitigate such problem by timely opening the circuit and properly setting the timeouts; however, this operation requires a careful tuning. On the contrary, synchronization helps reducing network traffic overload. Services do not send requests unless they have been notified the services they want to communicate with have become ready. This way, only one successful request is sent and the network traffic is significantly reduced. The same considerations can easily be generalized to the continuous deployment pipeline scenario.

In fact, in the second case, whenever synchronization features are not natively supported by the pipeline orchestrator or by the underlying microservices platform, they should be externally provided. Therefore, the framework could help the pipeline orchestrator suspend the testing meta-Pod until the building and the deploying meta-Pods have not built and deployed the application Pods yet, and these, in turn, have not entered their Ready state yet.

Finally, sychronization should help in improving the lead time by reducing its wait time component to its bare minimum. Figure 7 describes timing relationships between the different CD phases and the possible synchronizations violations and faults which they could incur.

According to Figure 7:

(a) in (a), no mechanism has been adopted; the testing phase starts soon after the deploying phase has been started, with a possible consequent violation and fault;
(b) in (b), a simple timeout mechanism has been adopted; as the timeout has been set too low, the testing phase starts before the deploying has been successfully completed and consequently leads to a violation and a fault;
(c) in (c), again a simple timeout mechanism has been adopted; since it is set too high, the testing phase starts some time after the deploying phase has successfully been completed; despite having no synchronization violation, this mechanism still does not comply with the goal of reducing lead time to its bare minimum;

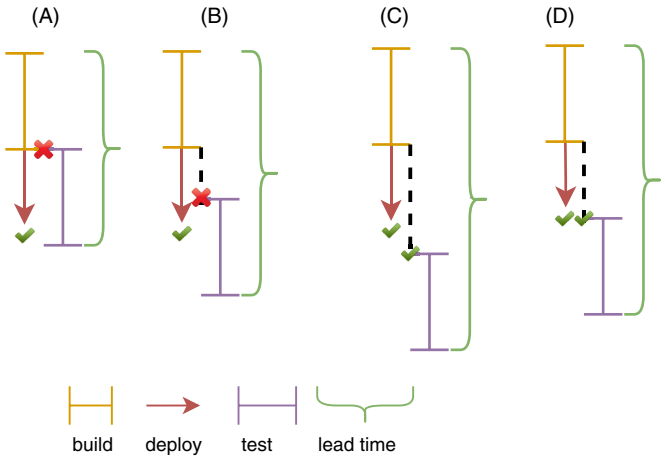

FIGURE 7 Lead time and violations [Colour figure can be viewed at wileyonlinelibrary.com]

(d) finally in (d), a more sophisticated synchronization mechanism has been adopted; as the testing phases start as soon as the deploying phase has successfully completed, there is no violation nor any lead time increase.

4.2 | Health state-based synchronization

According to the discussion about synchronization, we propose a framework based on a platform service called *Synchronizer* to implement synchronization in the two scenarios presented before. It works by exploiting health state probes and registry features to suspend dependent microservices execution until their dependencies have not become ready yet. It has also been designed to ensure extensibility in order to possibly deal with the other synchronization problem classes previously introduced.

In addition to service discovering features, modern registries also allow to keep track of state information related to microservices by implementing a persistence layer, usually through a key-value store, and by providing a set of RESTful APIs. Such APIs may directly be accessed through HTTP calls or may be wrapped in a client, which can later be used as a library by other software. The APIs may be complex but at least they all provide this basic set of operations:

- *Get*—to retrieve the value associated to a key
- *Put*—to update a key value, or add a new key value if it does not previously exist
- *List*—to list a set of keys according to some prefix
- *Watch*—to register a key and observe its value changes
- *Delete*—to delete a key

At the same time, Pod-based environments typically provide a set of probes, which can periodically check for microservices health state and consequently update the corresponding key values in the registry. The probes can either be built-in or can work on an application level and be customized according to each microservice business logic.

Figure 8 shows the way in which *Synchronizer* works both with an application bootstrapping (choreography) and with a continuous deployment pipeline (orchestration). In Figure 8A, *Synchronizer*, an external process, works by watching the registry for the microservices health state. However, it has now been containerized and its command line parameters can be passed in at execution time. Specifically, every time a Pod has some dependencies, it is equipped with *Synchronizer* in the form of an *init container*. In this way, the same container image can be reused for every Pod and only the init container's parameters must be changed.

Instead, in Figure 8B *Synchronizer* uses a registry client library to register a watch and observe microservices health state, namely by collecting values provided by the probes for each Pod constituent a microservice. *Synchronizer* is again an external process, which, wrapped in an adapter and provided with the requested parameters, can be invoked through a specific interface. In particular, the pipeline orchestrator invokes *Synchronizer* through the support of the underlying operating system, namely the *shell*. Despite the burden inherent in implementing a specific adapter, this approach provides *Synchronizer* with more flexibility of usage by decoupling its implementation from its invocation.

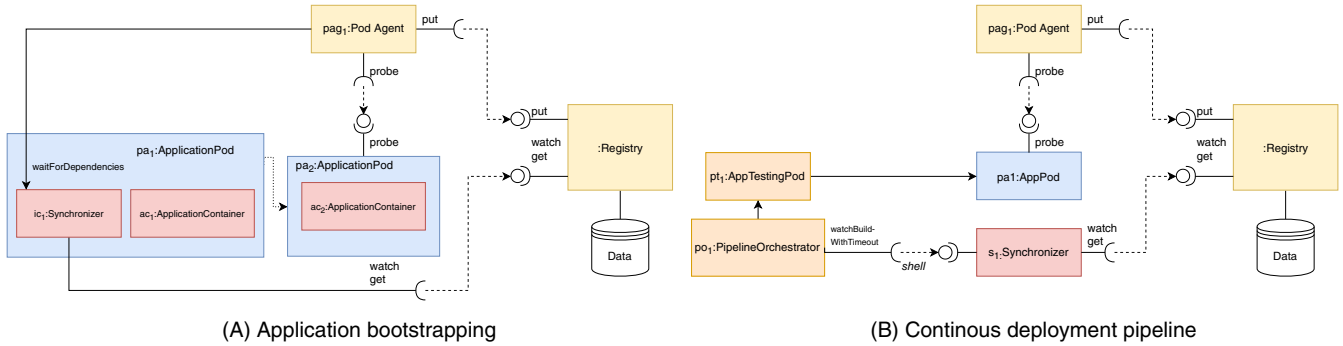

FIGURE 8 *Synchronizer* at work [Colour figure can be viewed at wileyonlinelibrary.com]

Exposing the same synchronization mechanism, by deriving specific conceptual primitives from each problem instance, improves *Synchronizer*'s conceptualization and usage. The primitive prototypes are listed below:

- `waitForDependencies (namespace string, dependencies string, timeout string, mode string)error`
- `watchBuildWithTimeout (namespace string, buildName string, timeout string, mode string)error`

The first parameter represents the *namespace*, to which all correspondent Pods and meta-Pods belong to. The only difference lies in the second parameter. For *waitForDependencies*, *dependencies* is a comma separated list of all the dependencies. Each element of the list, composed with the namespace prefix, allows for identifying the specific dependency and any other related resource. Instead, for *watchBuildWithTimeout*, composed with the *namespace* prefix, parameter *buildName* allows for identifying the building process and any other related resource. The third and the fourth parameter deal with non-functional requirements.

timeout is used to deal with byzantine faults. As typical distributed systems faults may disrupt pipeline execution, *Synchronizer* has a timeout mechanism, by which if no synchronization is observed in a preset time interval, then an error condition is raised.

mode, on the other hand, is used to denote how *Synchronizer* works. If a microservice has no *Service/Endpoint* abstraction or is composed by a single Pod, *mode* has no effect. However, if a microservice has multiple Pod replicas, then *mode* affects the way the synchronization is dealt with. Specifically, we have two modes:

- *all*—it is useful in all those situations where every Pod must be ready; for example, in the case of a performance test on a service;
- *atleastonce*—it is useful in all those situations where it suffices at least one Pod to be ready; for example, in case of a functional test. Also, this semantic increases the likelihood of success since synchronization does not happen on a single preset number of replicas, but, on the contrary, on any instance: in this way the likelihood of byzantine faults is reduced, assuming we are not interested in having all the replicas ready.

Finally, an *error* is returned in case of faults. However, when everything works correctly, *error* is set to null.

The general mechanism of *Synchronizer* is described in Figure 9. To begin with, *Synchronizer* needs the appropriate parameters to determine when and which Pods' execution it has to suspend. Such parameters, not shown here for brevity reasons, depend on the specific prototype. Since Pods are run asynchronously with respect to the underlying environment and to the registry, observation of value changes should start from a past moment in order not to lose valuable events; key-value stores can provide this feature by storing past key versions (the so called revisions) and persisting only their delta differences. Therefore, to determine when to start the observation, *Synchronizer* retrieves the *modRev*, that is, the appropriate modification revision. Then, as a Pod could be deployed in several replicas, *Synchronizer* also retrieves the *replicasNumber*. After this operation, *Synchronizer* registers a watch with prefix, that is, the dependency deployment name, to listen for Pods creation and when such event occurs, it saves the related Pod name. Having the Pod name,

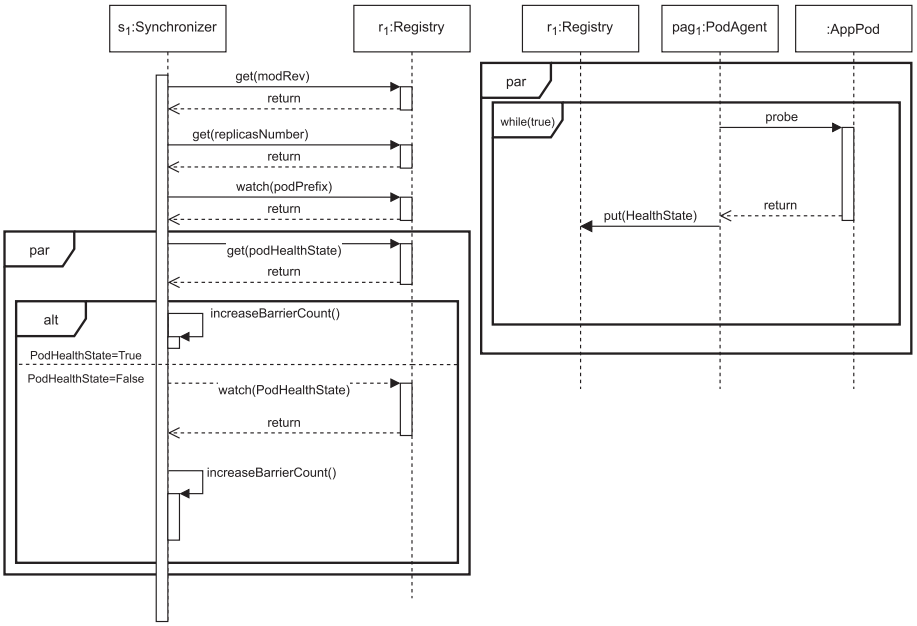

FIGURE 9 *Synchronizer* sequence diagram

Synchronizer can wait for the pod to be running. As soon as this happens, it can wait for the Pod to become ready. At this point, as registering a watch is a resource-consuming operation, *Synchronizer* makes use of an optimization and thus first retrieves the health state value to check whether it is already valid. Such optimization exploits the aforementioned asynchrony: by the moment *Synchronizer* arrives to this step of its execution, the health state may be already valid and it would be counter-productive to use a watch. However, if that is not the case, *Synchronizer* proceeds to register a watch for each Pod and to wait for the health state to become true; the watch is set to start from the Pod's modification revision as it would be useless to retrieve past changes because at the moment of *get* it was not ready yet. Finally, when every replica is ready, *Synchronizer* successfully returns the control back.

However, a Pod could depend on multiple Pods. In such a case, *Synchronizer* collects every dependency and creates a separate thread of control for each of them, while passing them the appropriate modification revisions. Each thread can then individually proceed as described above.

The mode influences the way concurrency is dealt with. In the case of mode *all*, as shown in Figure 9, the use of a barrier implies that *Synchronizer* does not return until all observed Pods have become ready. Instead, mode *atleastonce* is not shown in Figure 9 for the sake of brevity; however, its behavior can be easily described: by implementing a shared channel, whichever Pod, whose health state is observed valid first, is chosen as the ready replica and allows the pipeline to continue its run smoothly. A barrier is also used to deal with multiple dependencies. Specifically, a Pod is allowed to run only when each of its dependency has reached the outer barrier, that is, when each is up and running.

As shown in Figure 8, the general mechanism can then be combined with the specific problem *Synchronizer* is used for; this grants a certain degree of flexibility. In particular, new *Synchronizer* prototypes could be implemented for specific problems without changing the core of the general mechanism.

In conclusion, since *Synchronizer* collects health state values by following a push-approach (registering a watch) rather than by means of a mechanism based on timeouts and *Get* operations, violations and errors are avoided.

5 | SYNCHRONIZER VALIDATION

This section reports on the results of an empirical evaluation conducted to validate *Synchronizer*. Its main task is to ensure safe communication between microservices by solving inherent coordination issues. Such issues could, however, affect different aspects of a microservices application, as seen in Section 2. Therefore, the evaluation has been planned to answer the two following research questions:

- *R1: can Synchronizer guarantee the bootstrap order decided at design time?*
- *R2: can Synchronizer improve CD pipelines performances by reducing their lead times?*

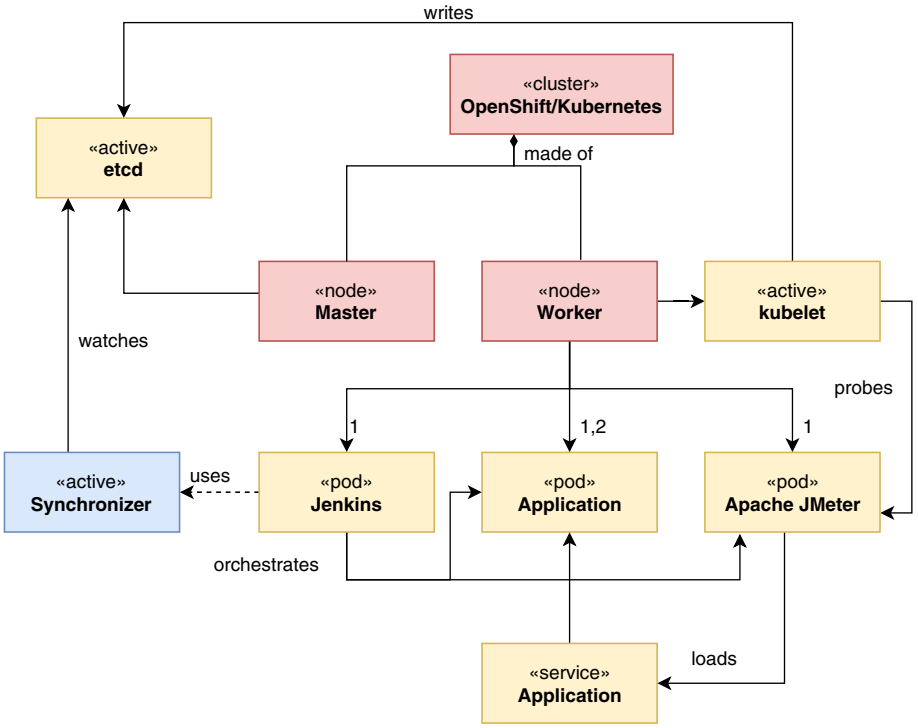

FIGURE 10 Testbed architecture [Colour figure can be viewed at wileyonlinelibrary.com]

Both validations have been performed on the OpenShift/Kubernetes platform, where we have implemented *Synchronizer* as a service. For *R1*, we have deployed a modified and unmodified *Sock Shop* version; whereas, for *R2*, we have implemented an ad-hoc continuous deployment pipeline[1]. In any case both tests represent a solid base to extend the support of the proposed synchronization framework to other classes of problems.

The methodology adopted for testing *Synchronizer* differs according to the peculiarities of the two scenarios we consider for the validation. Therefore, the details of the adopted methodology are reported in the specific subsections dedicated to the in-depth description and discussion of the tests. Since the scenarios and *Synchronizer* itself depend on the underlying platform, we first give a high-level description of the platform, then we describe the tests and the related results to answer the research questions. Finally, the results are discussed with the aim of highlighting possible threats to *Synchronizer* validation.

5.1 | Target infrastructure

A high-level representation of the target architecture for both the scenarios is presented in Figure 10. In order to keep the representation simple and focused, only the necessary components of the OpenShift/Kubernetes platform are displayed. Moreover, in Figure 10B, for the sake of simplicity, we have shown just a subset of the application under test.

OpenShift[22] (Origin, community open source v3.9.0+ba7faec-1) is an application container platform to build, deploy, serve, and orchestrate containers. It uses Kubernetes as its orchestrator and Docker as its container engine. Docker provides the abstractions to create and package container images; whereas Kubernetes orchestrates the containers. On top of this, OpenShift efficiently builds and deploys applications and scales them accordingly.

Kubernetes[21] (v1.9.1+a0ce1bc657) is an open source orchestrator for deploying containerized applications on multiple hosts and for their maintenance and scaling. Kubernetes adopts a master-slave architecture and uses *etcd* for service discovering and configuration data. *etcd*[24] (version 3.2.22) is an open source key-value store to reliably store data inside a cluster. Interestingly, *etcd* provides powerful APIs to work with its content: the Watcher API implements an event-based interface to asynchronously monitor key changes.

Kubernetes implements the concept of Pod as its smallest deployable unit and the *Service/Endpoint* abstraction by means of *Service*. The Pod life cycle can be summed up by looking at the *PodStatus* field. This has a subfield *phase*, which

[1] Available at https://github.com/woland7/microservices-synchronizer-config.

indicates where the Pod is in its life cycle and can take on the following values: *Pending*, *Running*, *Succeeded*, *Failed*, and *Unknown*. Moreover, *PodStatus* has an array of subfields, called *PodConditions*, which provides further information on the Pod life-cycle. The *PodConditions* are the following: *PodScheduled*, *Ready*, *Initialized*, and *Unschedulable*; their status can either be *True*, *False*, or *Unknown*. For a Pod, Kubernetes also implements the concept of init containers: if they do not successfully terminate, the application containers are not run. Finally, a failed Pod may be automatically restarted according to its *restartPolicy*, which may be set to *Always*, *OnFailure*, and *Never*.

The *kubelet* agent, executing on each worker node, ensures every Pod is running correctly and keeps track of changes by writing them on *etcd*. Moreover, the *kubelet* probes the Pods to check their health. Probes help identify errors such as when the container inside a Pod is stuck in a deadlock. This diagnostic component is periodically executed by the *kubelet* to monitor the state of the microservice: it works by calling a handler implemented by the container. A probe comes in two forms: *readiness* and *liveness* and both probes can be defined for each constituent container. When a HTTPGetAction handler is used, the *kubelet* executes a HTTP Get request towards the specified container's IP address and a specific path. The diagnostic succeeds if the status code is $\geq$200 and 400. The readiness probe indicates if the container is ready to serve requests: if the probe fails for at least one container, then the Pod's IP address is removed from all the services that match the failing Pod and its *PodCondition Ready* is set to *False*. As shown in Figure 10, the *kubelet* writes the probe result on *etcd* by setting the appropriate field for the specific Pod.

To set up the testing infrastructure for the application bootstrapping, we have deployed *Sock Shop* twice. In the first case, it has not been modified: each microservice is described by a deployment and by its service. In the second case (see Figure 10B), instead, each microservice has been equipped with a *Synchronizer* init container to correctly deal with the dependencies.

On the other hand, to set up the testing infrastructure for the continuous deployment pipeline, three Pods have been created on the worker node: Jenkins as the pipeline orchestrator, JMeter for testing and a Web application.

Jenkins[25] (ver. 2.89.4) is a CI/CD automation server adopting the pipeline-as-a-code principle and orchestrates both the application and testing Pods. Pipeline's code is written in a Jenkinsfile through its own DSL language, derived from Groovy. In any case, not all the code should be kept in Jenkinsfiles; reused code can be factored out and kept as a shared library. Moreover, Jenkins provides several plugins; in particular, the OpenShift Pipeline Jenkins Plugin[26] allows to build and deploy applications. *Apache JMeter*[27] (version 5.0) is an open source software Java application designed to test functional behavior and performance of an application under load. As we are simply interested in evaluating whether *Synchronizer* is able to keep the pipeline running smoothly, we have decided to keep the application as simple as possible. Therefore, the Web application, deployed in either 1 or 2 replicas of Wildfly Pods, implements a compute intensive function based on servlets and hosts a probe for observing the state. The probe responds with the HTTP Status Code 200; instead, the servlet simulates a computing intensive application. As the application Pod completes its operations to get fully initialized, it soon advertises its health state as ready; therefore, we can determine whether *Synchronizer* blocks the testing meta-Pod from sending requests before the application Pod becomes ready; on the other hand, by using a computing-intensive application, we can evaluate how the whole pipeline reacts to higher workloads according to each deployment strategy and thus get useful insights to further improve *Synchronizer*.

OpenShift provides two deployment strategies: *Recreate* and *Rolling*[2]. The former provides the most basic form of deployment. When an application is first deployed, Pods are concurrently deployed from 0 up to the number of replicas. If a version of the application already exists and a new deployment is initiated, the existing Pods are scaled down completely to 0 first and then a roll-out pattern similar to the initial deployment follows. Since Pods containing the previous version are quickly replaced by the new version, *Recreate* is ideal during the application development phase: the ability to quickly test and validate the latest changes reduces the overall cycle time for a development team. On the contrary, the Rolling strategy better suits productions objectives where downtimes are not permitted: when a new roll-out is executed, an old replica keeps being active; mission critical applications that have a high service level agreement should use this strategy. It guarantees application uptime by providing a finer control over the deployment process. Instead of scaling down existing instances of the application before deploying the new version, instances of the new version of the application are instantiated over time. Only once a replica of the new version becomes active a previous version instance is removed.

The OpenShift/Kubernetes architecture in Figure 10 has been deployed on three nodes, each composed of a virtual machine hosted on an OpenStack cluster. The specific nodes are shown in Table 1. All the control components and *etcd* have been installed on node *master*; *infra* keeps all the infrastructural components, such as the *Docker registry*; and *node* has been reserved for deploying applications.

[2]There is also a Custom strategy but it has not been considered in this work.

TABLE 1 OpenShift nodes specifics

hostname	CPU@3.19MHz	RAM	Disk
master	6	8 GB	35 GB
infra	4	4 GB	20 GB
node	4	4 GB	20 GB

Synchronizer[3], developed in Go, implements the synchronization by exploiting the readiness probes and by communicating at a lower level with *etcd*. It suspends microservices execution until its dependencies have become ready. A Pod is ready and able to receive traffic if it is Running and has passed its readiness probe. However, communicating with *etcd* may also allow to watch for other conditions and extend the *Synchronizer*'s application scenarios.

In Kubernetes, each *kubelet* writes the *PodCondition Ready* value on *etcd*; therefore, the synchronization is implemented by using for each Pod an *etcd* Watcher listening to the desired state change. If the specified mode is *all*, each Pod's *PodCondition Ready* value should be set to True to achieve the synchronization; instead, if the mode is *atleastonce*, again an *etcd* Watcher is created for each Pod, but the synchronization occurs as soon as at least one Pod's *PodCondition Ready* value has been set to True. A timeout mechanism has been added to handle byzantine faults: the Watchers do not hang on indefinitely but *Synchronizer* returns an error if the timeout has passed. As explained in Section 4, *Synchronizer* can be executed through the shell by means of an adapter; this has been implemented through a Jenkins shared library for the continuous deployment pipeline[4].

Synchronizer's interactions with the OpenShift components are shown in Figure 10. In particular (Figure 10A), for the application bootstrapping, *Synchronizer* is included in an init container and blocks the execution of a microservice until the services it depends on have become ready. Instead for the continuous deployment pipeline, in (Figure 10B), *Synchronizer* suspends the pipeline execution so that the Apache JMeter Pod is deployed and run only when the Web application Pods have become ready.

5.2 | Application bootstrap (*R1*)

In this subsection, we aim at answering research question R1. To this end, we have exploited the linearization properties of *etcd*. In fact, *etcd* guarantees linearizability by default for write operations. Therefore, since the last update to a Pod represents the moment it has become ready and running, we have launched the application with and without *Synchronizer* and then checked whether the order of updates follows the dependencies relationships.

Listing 1 shows the last revision for each Pod and highlights how the order is unconstrained and incorrect as the last Pod to become ready is *user-db*, despite *user* depending on it. Instead, in Listing 2, *Synchronizer* constrains the microservices start-up order to the one illustrated in Figure 1.

```
orders-755bd9f786-zdvtz;7563625
catalogue-db-6794f65f5d-zpjn8;7563638
carts-6cd457d86c-gwjxw;7563642
front-end-679d7bcb77-vv9cl;7563653
catalogue-779cd58f9b-9nd2t;7563656
queue-master-5f98bbd67-cm9sc;7563681
carts-db-784446fdd6-xqwct;7563686
shipping-79786fb956-5b524;7563690
user-6995984547-cntjq;7563693
payment-674658f686-rxn6m;7563702
orders-db-84bb8f48d6-xgc7f;7563708
rabbitmq-86d44dd846-5xzj5;7563709
user-db-fc7b47fb9-25p2n;7563762
```

Listing 1: Update timestamps without *Synchronizer*

[3] Available at https://github.com/woland7/synchronizer.
[4] Available at https://github.com/woland7/jenkins-synchronizer-sharedlibrary.

```
catalogue-db-6794f65f5d-xnwkk;7564745
catalogue-54789786d-wdpf2;7564770
carts-db-784446fdd6-x95bv;7564783
queue-master-5f98bbd67-r2h7f;7564790
rabbitmq-56ddbf4f7d-wd8tg;7564809
payment-84798b58fd-fwwfn;7564817
orders-db-84bb8f48d6-qx84m;7564824
shipping-7d9b5b6c86-bdhlt;7564829
carts-588779c777-c6ck9;7564833
orders-5865b9546b-7t454;7564845
user-db-fc7b47fb9-r2f57;7564889
user-fb6cc7d69-g8n59;7564899
front-end-5cbb586d8f-bvrjj;7564911
```

Listing 2: Update timestamps with *Synchronizer*

The repetition of the tests several times always produced consistent results. In particular, the absolute order may change because of the scheduling, but the relative order the microservices start up was not affected.

To show that *Synchronizer* also helps reducing network traffic overload, we ran another set of experiments. We containerized *JMeter* and added it as a new service to the application, with a single dependency on *front-end*. This needs internal services to serve the requests it is charged with; therefore, the bootstrap order would impact the application performances, when one or all of the services involved in a request are not ready yet. For this reason, we have identified an endpoint which meets our requirements. Specifically, we have chosen an endpoint[5] used to get a catalogue product information: *front-end* has to contact *catalogue*, which in turn has to contact *catalogue-db*. Then, we have set up two configurations, each of 25 runs of 10 consecutive requests towards this endpoint: one *(i)*, where *Synchronizer* handles every dependency and the other *(ii)*, where *Synchronizer* has been removed from *front-end*, and thus the latter starts up without waiting for the internal services, introducing the probability of getting a *503 Internal Server Error*. In particular, in *(ii)* we aimed at measuring the impact of the retry time interval on the network traffic overload. For this reason, we set such interval to the barest minimum: each request is sent after the other in a burst; in this way we minimized the synchronization error while maximizing the amount of retries. In *(i)*, as expected, no error has been encountered; in *(ii)*, however, at least an error has been encountered in 84% of the runs. In particular, in 12 of these cases, the error has been encountered at least five times. This means that at least five out of 10 requests failed: therefore, when *catalogue* is not ready yet, at least 10 useless messages flood the network. However, were Synchronizer not used at all, the worst case would have occurred: the innermost service, catalogue-db, would have not been ready yet and at least 15 useless messages would have flooded the network. In addition to that, if services took even more to start-up and become ready, the amount of useless messages would increase even further.

5.3 | Continuous deployment pipeline (*R2*)

In this subsection, we aim at answering research question R2. In fact synchronization violations could induce higher lead times in scenarios involving continuous deployment pipelines. Therefore, with this experiment we verified whether *Synchronizer* helped reducing the wait-time, that is, the dead time between two stages and consequently the entire lead time. The tests could have been configured to directly measure the reduction and the synchronized pipeline could have been compared with a timeout-based pipeline, that is, a pipeline whose delays between stages are preset to avoid synchronization violations. However, this approach has some drawbacks. First, an asynchronous system has no upper bound for delays, therefore the timeout should have been set very high, theoretically infinite to ensure there would not be any violations. Second, the timeout values would have been highly dependent on the load on the system. For these reasons, R2 has been reformulated as follows:

- *R2.1, can we dynamically constrain a pipeline wait time based on the SUT state, without falling into errors?*

[5] The internal endpoint is http://front-end.sock-shop.svc/catalogue/03fef6ac-1896-4ce8-bd69-b798f85c6e0b.

Therefore, a different approach has been taken: the synchronized pipeline has been compared with a plain pipeline where no synchronization mechanism is used by evaluating the impact on the lead time through the observation of the synchronization violations. Moreover, by adopting this approach, we derive a metric that focuses more on the framework itself than on the specific scenario; therefore, it has a wider scope and can serve as a basis to estimate the general effectiveness of the synchronization mechanism behind our framework.

In particular, the whole pipeline has been tested by submitting a synthetic load generated by JMeter to the Web application, which acts as the SUT. The test configurations vary according to the following parameters: the deployment strategy (Recreate|Rolling), the replicas numbers (1|2) and when the replicas number is set to 2, the mode (ALL|ATLEASTONCE). Moreover, we overloaded the environment by generating a total of 256 requests from a variable number of 2, 4, or 8 threads activated in 5 seconds (rump-up period) in order to generate a high degree of concurrency.

The following figures show the test results for each configuration. Ordinates axes have been scaled logarithmically to provide a more compact view. Only meaningful results are shown. The possible error conditions are the following:

- *Not Synched* (NS)—if the requests are sent before the Pods become ready, they find the No Route To Host Java exception; consequently there is no synchronization and there is a violation;
- *Overloaded* (OL)—if too many concurrent requests are sent to the application, this may stop working and again the requests find the No Route To Host Java exception; however, this case does not represent a synchronization violation and thus it is tagged as OL;
- *Not Found* (NF)—when the Rolling strategy is used, old and new application versions might be running together at the same time; in this case, some requests may be served by an old replica that returns a 404 *Not Found* error, which is also considered as a violation.

5.3.1 | Recreate 1 replica

The first set of histograms in Figure 11 shows the test results when the *Recreate* strategy is used and only one replica is deployed. In particular, Figure 11A refers to the case where no synchronization mechanism is used. As expected, the *Not Synched* error is encountered for any threads configuration. Moreover, as the threads number increases, that is as the number of concurrent requests increases, latencies tend to become more equally distributed over a wider interval. However, at the same time, as the total amount of requests is fixed, the *Not Synched* error also grows with the thread numbers because the application Pod receives more requests in the same time interval. On the contrary, with a lower threads number, the application Pod receives requests at a lower rate and consequently, when it becomes ready, it has to receive many more requests yet.

Figure 11B refers to the case when *Synchronizer* is used. The results are as expected for any threads configuration: that is, no error is encountered because no request is sent before the application Pod becomes ready. However, four and eight threads configurations are affected by an overloading problem; since the application Pod receives requests only when

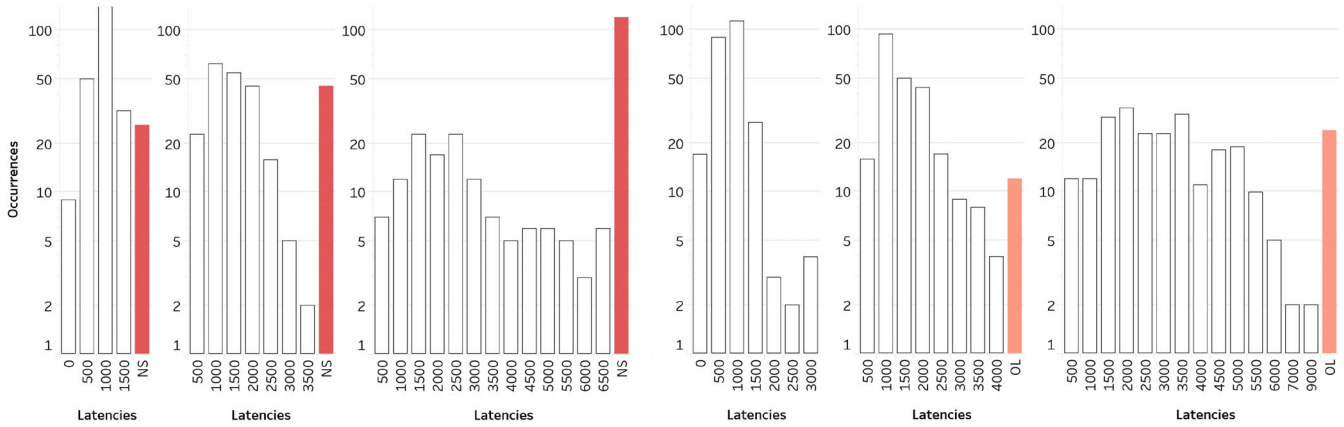

FIGURE 11 Deployment strategy: recreate; number of replicas: 1 [Colour figure can be viewed at wileyonlinelibrary.com]

ready, the amount of concurrent requests is even higher than the previous case and thus, the application Pod goes out of resources and the *Overloaded* condition occurs.

5.3.2 | Recreate 2 replicas

The diagrams in Figure 12 show the test results with the *Recreate* strategy when two replicas are deployed. Figure 12A, in particular, displays the distribution of latencies for this configuration when *Synchronizer* is not used. Despite lower latencies are expected due to the load balancing guaranteed by the two replicas, again there is an increase of the *Not Synched* errors as the threads number increases. Figures 12B,C show test results when the synchronization mechanism is used, respectively with *all* and *atleastonce* modes. In particular, there is no substantial difference in latencies distribution between the two modes; in fact, slight differences might be ascribed to fluctuations in the inner mechanisms of OpenShift and, as every Pod has been deployed on the same node, to its underlying hardware. However, despite not encountering it, we argue that in such configuration the *atleastonce* mode might be affected by the *Overloaded* condition. In fact, according to its semantics, the synchronization occurs as soon as a replica becomes ready. If the other replica does not become ready on time or does not become ready at all, an overloading problem may be encountered. For this reason, such mode might be a better fit for a functional test rather than for a performance test.

5.3.3 | Rolling 1 replica

The diagrams in Figure 13 show the test results when the *Rolling* strategy is used and only one replica is deployed. From the test perspective, this strategy does not differ from the *Recreate* strategy; in particular, in Figure 13A, when the pipeline

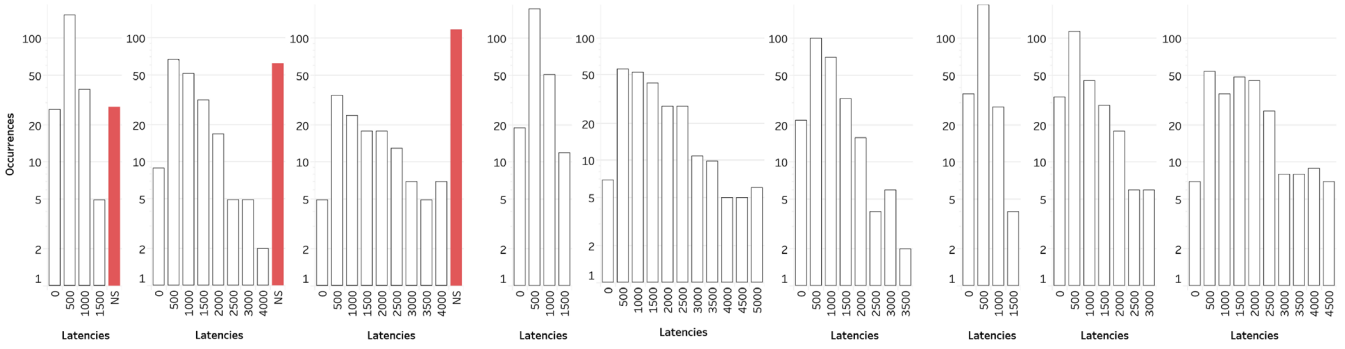

FIGURE 12 Deployment strategy: recreate; number of replicas: 2 [Colour figure can be viewed at wileyonlinelibrary.com]

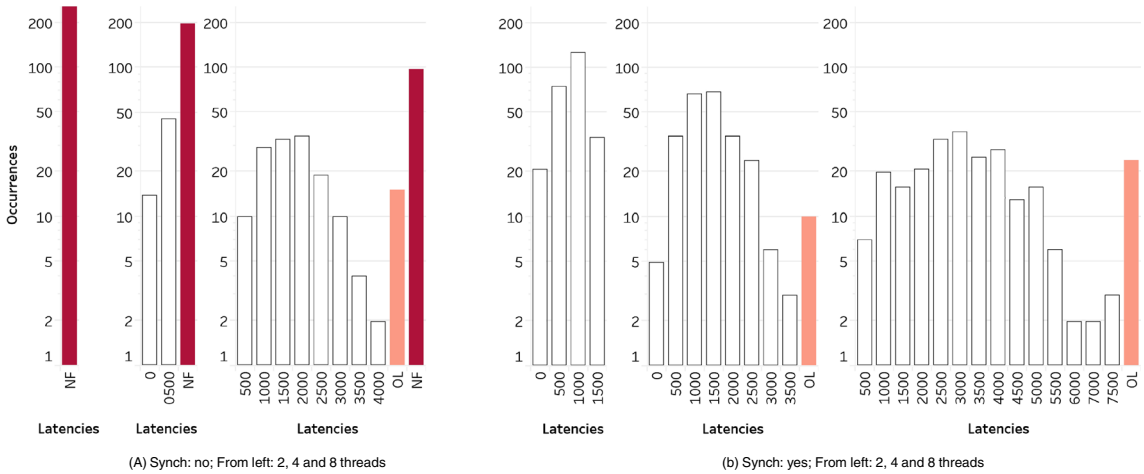

FIGURE 13 Deployment strategy: rolling; number of replicas: 1 [Colour figure can be viewed at wileyonlinelibrary.com]

orchestrator does not use *Synchronizer*, errors occur as expected. However, the errors we encounter differ from those of Section 5.3.1. In fact, we argue that with the specific application under test, the inner mechanism of the *Rolling* strategy has a higher impact on the underlying hardware with respect to those of the *Recreate* strategy. In particular, when the pipeline orchestrator starts the testing meta-Pod, the application Pod has not become *Running* yet. As a consequence of this, requests are momentarily served by the old replica, whose endpoint has not been updated, and they are met with the *Not Found* error. With eight threads configuration, however, some requests are served correctly; we argue it happens because the tests take more to start up and meanwhile the old replica has already been replaced with the new one. Instead, as displayed in Figure 13B, the use of the synchronization mechanism implies that requests are served correctly. However, as with the *Recreate* strategy, when the number of threads increases, we are met with the *Overloaded* condition; slight differences in latencies distribution are again due to inner fluctuations in the mechanisms of the two strategies.

5.3.4 | Rolling 2 replicas

The case involving the *Rolling* strategy and the deployment of two replicas is the most critical. In Figure 14A, the lack of the synchronization mechanism implies the same result as the corresponding case with one replica; however, as usual, a better distribution in latencies is due to the deployment of two replicas. In Figure 14B, *Synchronizer* is used with mode *all*: as expected, the requests are served correctly with no overloading problem; therefore, from the test perspective, this case can be assimilated to that of Figure 12B, with slight differences in the distribution because of the inner mechanisms. Finally, Figure 14B shows the only case when *Synchronizer* fails, that is when the *Rolling* strategy is combined with the *atleastonce* mode; however, this is not an implementation problem and will be discussed in Section 5.4.

5.4 | Threats to validity

Even though the current implementation of *Synchronizer* is sufficiently robust and the experimentation demonstrated its validity in some typical scenarios for microservices applications, some threats to validity require a further discussion.

In case of the *Rolling* strategy with two replicas and mode *atleastonce*, the synchronization occurs as soon as at least a replica is ready, thus for a certain amount of time both the old and the new replicas are ready and running. For this reason, some requests may be served by the old replica and are met with the Not Found error (see Figure 14C). This subtle error condition and consequent violation happens only with two and four threads. Since the eight threads configuration requires more resources for the test bootstrap phase, such phase takes longer and actually terminates when even the remaining replica becomes ready. However, a careful examination of this problem revealed that it does not depend on *Synchronizer* but rather resides in the mechanism behind the Rolling strategy; therefore, we cannot improve *Synchronizer* to deal with this issue.

Another aspect that could influence experimental results is system overloading. If tests were to be repeated with a higher or lower load, the latencies variations may differ as well as the amounts of synchronization violations and other

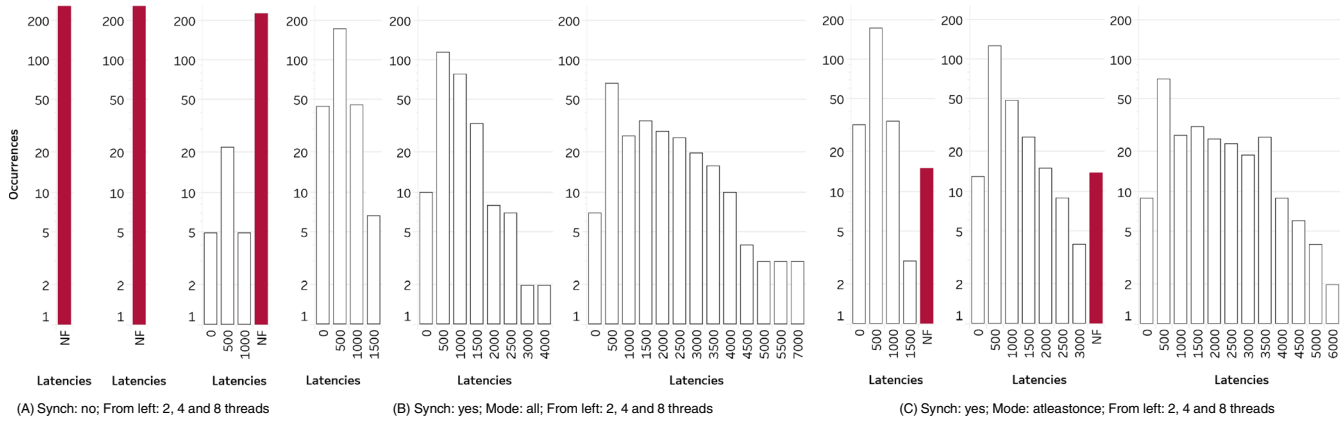

FIGURE 14 Deployment strategy: rolling; number of replicas: 2 [Colour figure can be viewed at wileyonlinelibrary.com]

error conditions. This does not hinder the functional aspects of *Synchronizer*, which works regardless of the load on the system. Yet, if the overload overcomes a critical threshold, then fewer resources may be available to *Synchronizer*, which could therefore possibly run slower with respect to the rest of the system. A slow run of *Synchronizer*, however, is not a problem in itself: violations would still be avoided but in a slower way because of the degraded performances. Yet, if the timeout were not adequately set to counterbalance such degraded performances, it might still be met causing *Synchronizer* to return an error, even though no functional problems are present. At the moment, the timeout is tuned manually, but a more sophisticated mechanism could be implemented in future work to deal with the aforementioned issue.

Finally, the way *Synchronizer* has been conceived implies that the registry must be well functioning, since the registry is at the core of the platforms upon which *Synchronizer* can be used. Therefore, even if the registry is external to *Synchronizer* and provided by the hosting platform, decentralized solutions for this critical component could ensure higher robustness to microservices applications.

6 | RELATED WORK

Choreography and orchestration are the main alternatives to coordinate communication in a distributed system. Both stem from the context of SOA (Service Oriented Architecture) and have valuable standard reference languages, WS-CDL and WS-BPEL, respectively. However, the Web services approach cannot be fully reused as microservices emphasize dumb pipes and smart endpoints,[28] which contrasts with the use of the complex aforementioned languages.

In any case, choreography is the general preferred[4] coordination model for microservices since decentralization represents a key point in this architecture; however, it is not yet fully mature[29] to be exploited in production environments. Besides, some phases of a microservices application life-cycle may also benefit from orchestration. In fact, RedHat has integrated Jenkins pipelines in its OpenShift platform to orchestrate system process flows, such as promoting applications in a continuous deployment fashion.[22]

Synchronizer can be positioned in the specific context of coordinating microservices (distributed entities) by synchronizing them on the basis of their running states, which we achieve by exploiting and extending novel features of service discovery systems.[30,31] Modern discovery services go further than just supporting discovery capabilities as they allow to retrieve state information about the services being discovered at different levels of granularity. Such capability, often supported through APIs, can be as much simple as getting state information or be more sophisticated and involve performing health checks or implanting a watch to listen for specific state changes. Therefore, two or more services can be synchronized or somehow coordinated by brokering the service registry of the discovery system.

Among the most popular discovery systems, Zookeeper is a high-performance coordination service for distributed applications. It exposes common services for naming, configuration management, and synchronization. The Zookeeper's synchronization mechanism is limited as the update notification does not carry the updated values and it suffers of some inefficiencies.[32] Consul,[33] based on an implementation of the Serf[34] algorithm, combines service discovery features along with distributed configuration management. It embeds a strongly consistent key-value store, robust monitoring, and health checking. However, Consul is not tightly integrated with Docker. Etcd[24] is the service discovery system used by the CoreOS project. Unlike Consul, it mainly supports distributed configuration management and it is based on an implementation of the Raft[35] algorithm. It does not support out-of-the-box health checking features. However, when it is integrated in CoreOS or container orchestrators, such as Kubernetes,[21] etcd is able to provide also service discovery. Moreover, it implements an interesting API, which allows to retrieve or listen for state information on the services in an efficient and scalable fashion. Unlike Consul and Etcd, Zookeper protocols are not built-in and thus etcd can be seen an enhancement on Zookeeper.[36,37] Finally, Serfnode[38] uses pre-loaded Serf agent containers on top of which applications can be loaded and run. It creates a network of Serfnodes that constitutes a fully decentralized light-weight, service discovery systems at the cost of lacking more complex features, such as state information retrieval.

The lack of a coordination mechanism in microservices interactions could be partially supported by some patterns. Circuit breaker appears to be a well suited solution to deal with communication issues and to prevent cascading failures. It is a pattern first popularized by Nygard[5] to prevent unanticipated events and enhance resilience in the context of enterprise systems. Specifically, it is aimed at dealing with network-related failures, as in when a target service answers too slowly or returns too often a fault. In particular, calls to the target service are wrapped in by another component, which keeps track of different parameters and trips the circuit when such parameters overcome their thresholds. By doing so, microservices communication problems can be mitigated as the circuit breaker prevents errors propagation by returning to the caller without performing any action. As a matter of fact, several works[39,40] suggest the adoption of circuit breakers in

developing microservices-based applications and solutions have already been implemented in practice. For example, Kubernetes implements a rather simple circuit-breaking mechanism by unhooking failed Pods from their matching services.[21]

However, both service discovery systems and the circuit breaker pattern do not fully solve coordination and synchronization problems among microservices. Service discovery systems allow to retrieve microservices state but usually do not support fully equipped synchronization mechanisms in their APIs. On the other side, circuit breaker allows to prevent cascading failures but cannot ensure a proper synchronization since at least one call to the wrapper should always be made no matter if the breaker is client, server or proxy-based. Moreover, for the same reason, it does not optimize resource usage and RTT ti as a listener-based solution would. Therefore, the push-based approach of *Synchronizer* would solve both problems and could be seen as a simpler, alternative and more effective solution than the circuit breaker pattern. In fact, *Synchronizer* introduces two main advantages. First, it would achieve synchronization by avoiding any useless interactions, which even though would not affect a suspended microservice's performance, would still overload the network traffic. Second, it would not require a dependent microservice to implement a specific response output when its dependencies are not ready yet. However, since currently *Synchronizer* solves synchronization problems at start-up time, a circuit breaker could be still useful for handling faults during the execution of an application.

For coordinating a continuous deployment pipeline, we have examined other orchestration platforms besides Jenkins and have discovered that the there is no native thorough support for implementation but it is either limited or left to further customization, and could, therefore, be covered by *Synchronizer*. In particular, both Azure and Spinnaker provide a system, by which expressions are evaluated to verify if certain in-between stages conditions are met,[41,42] but do not provide a built-in function for synchronization. In addition, Spinnaker also implements support for custom stages, which may, through external tools,[42] provide the needed synchronization. Finally, custom stages and external tools are also supported by Jenkins[25]; in fact, its architecture makes a wide use of external tools while having Jenkins and Kubernetes at its basis.[43]

The problem of complex synchronization among microservices is still in its embryonic phase as demonstrated by the scarcity of papers addressing the problem. To the best of our knowledge, Pipekit[44] represents a first attempt in this direction where the authors follow a different approach than ours; they aim at tackling the synchronization problem by extending *docker-compose* with new directives which would allow to govern the deployment of microservices in a synchronized order; however, the work is still a proof of concept and has yet to be actually implemented. Their solution may also lack interoperability since it may be confined to the Docker ecosystem; on the contrary, the framework *Synchronizer* is based on allows to use it with different adapters and also to further extend it to support other scenarios.

To wrap the synchronization code, the *Sidecar* design pattern has been used. Specifically, it extends and enhances a main container: for example the main container, a Web server, might be paired with a logsaver sidecar container that collects the Web server logs from local disk and streams them to a cluster storage system. By using an additional container and communicating through *localhost* rather than embedding the new functionality in the same container, one achieves several benefits in terms of resource accounting and allocation, packaging, reuse, deployment, and failure containment boundary.[45] Therefore, by containerizing *Synchronizer*, we provide microservices that need to deal with synchronization issue with such sidecar. In this way, we enforce the fundamental principle of simplicity at the heart of the microservices architecture by keeping the synchronization code separated from the application code. In turn, this improves the possibility of reuse of *Synchronizer* as it would simply be reduced to attach a sidecar to the needed microservices. Finally, even the implementation of model-driven design tools would be simplified, as the generated code would be affected only by the dependencies.

7 | CONCLUSION AND FUTURE WORK

This article has addressed a new problem in microservices paradigm: the need for coordination by synchronization among microservices in order to avoid useless or faulty interactions. We have proposed a general framework for microservices synchronization, named *Synchronizer*, a new platform service that is able to synchronize microservices interactions on the basis of Pods health states. The framework has been implemented and tested within a specific technological stack, that is, the OpenShift/Kubernetes platform and validated against an application bootstrap and a continuous deployment pipeline.

Synchronizer can be considered as an extension of current discovery systems to support coordination among microservices on the basis of their running states. It has been purposely defined to support different usage scenarios in

microservices applications development. Therefore, the high level primitives that we provide have been thought specifically for deployment and execution units (such as Pods) which are the basis for running scalable and resilient applications in modern hosting infrastructures for cloud computing.

Through the discussion of specific and fundamental scenarios, the article also highlights the importance of coordination based on the running states of microservices for designing and running robust applications. Moreover, the two scenarios gave us valuable feedback to assess the effectiveness of the approach and ideas for future enhancements. Specifically, by using *Synchronizer*, the application bootstrapping scenario showed that our service is able to constrain an ordered start-up of the constituent microservices of an application. On the other hand, continuous deployment pipeline synchronization violations have been avoided and the corresponding lead time has been minimized.

The examples have proved the flexibility of *Synchronizer*, since it has been used both in a choreographed solution (as during an application bootstrapping) and in an orchestrated one (as in a continuous deployment pipeline). It is worth to note that the two examples can also be merged, as in developing a complex application by using DevOps practices and tools: once *Synchronizer* ensures that every constituent microservice is started after its dependencies are ready, it can also provide support to begin the testing only when the whole application is ready.

The test results provide us with a good basis to further extend the application scope of our framework to other synchronization problems. For example, in the context of developing a simpler alternative solution to the circuit breaker pattern, *Synchronizer* could be used to coordinate the interactions at execution time. In particular, by further harnessing registry and probes features and by enhancing the use of the sidecar design pattern, a new research direction may involve a decentralized approach where *Synchronizer* could help mediate interactions between Pods at execution time so as to forbid interactions when a critical service goes abruptly down or to help during scheduled maintenance windows. Finally, the study of these scenarios suggests to go further by supporting greenfield design with model-driven tools in order to automatically generate the needed synchronization code from services dependencies. On the other hand, in the context of brownfield design, reverse engineering tools for extracting dependencies could be useful to generate the supporting code for synchronization.

ACKNOWLEDGEMENTS

This work has been supported by the MIUR PRIN GAUSS and SISMA projects. We also thank TIM S.p.A. for having supported this work by providing us the data center we used to run OpenShift with *Synchronizer*.

ORCID


Eugenio Zimeo https://orcid.org/0000-0003-4683-5487

REFERENCES

1. Evans E. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Boston, MA: Addison-Wesley Longman Publishing Co., Inc; 2003.
2. Fowler M. Microservices—a definition of this new architectural term; 2014. https://martinfowler.com/articles/microservices.html. Accessed 2018.
3. Newman S. *Building Microservices: Designing Fine-Grained Systems*. Sebastopol, CA: O'Reilly Media, Inc; 2015.
4. Dragoni N, Giallorenzo S, Lafuente AL, et al. *Microservices: Yesterday, Today, and Tomorrow*. New York, NY: Springer; 2017:195-216.
5. Nygard MT. *Release It! Design and Deploy Production-Ready Software*. 2nd ed. Pragmatic Bookshelf: Raleigh, NC; 2018.
6. Esparrachiari S, Reilly T, Rentz A. Tracking and controlling microservice dependencies. *Queue*. 2018;16(4):44-65.
7. Aderaldo CM, Mendonça NC, Pahl C, Jamshidi P. *Benchmark Requirements for Microservices Architecture Research*. Buenos Aires, Argentina: IEEE Press; 2017:8-13.
8. Zimmermann O. Microservices tenets. *Comput Sci-Res Develop*. 2017;32(3-4):301-310.
9. Fowler M. *Continuous Integration*. ThoughtWorks; 2014. http://www.martinfowler.com/articles/continuousIntegration.html. Accessed 2018.
10. Pittet S. Continuous integration vs. continuous delivery vs. continuous deployment; 2018. Accessed 2018.
11. Shahin M, Babar MA, Zhu L. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access*. 2017;5:3909-3943.
12. Lwakatare LE, Kuvaja P, Oivo M. *Relationship of DevOps to Agile, Lean and Continuous Deployment*. Cham: Springer International Publishing; 2016:399-415.
13. Farcic V. *The DevOps 2.0 Toolkit: Automating the Continuous Deployment Pipeline with Containerized Microservices*. 1st ed. Scotts Valley, California: CreateSpace Independent Publishing Platform; 2016.
14. Kupiainen E, Mäntylä MV, Itkonen J. Why are industrial agile teams using metrics and how do they use them? *WETSoM 2014*. New York, NY: ACM; 2014:23-29.

15. Kupiainen E, Mäntylä MV, Itkonen J. Using metrics in agile and lean software development–a systematic literature review of industrial studies. *Inf Softw Technol*. 2015;62:143-163.
16. 7 Metrics to track when implementing continuous delivery; 2018. https://www.datical.com/blog/7-metrics-to-track-when-implementing-continuous-delivery. Accessed 2018.
17. Michael B. Measurements towards continuous delivery; 2015. http://www.agileadvice.com/2015/04/22/agileengineering/measurements-towards-continuous-delivery. Accessed 2018.
18. Dan Z. TechTack: the continuous delivery metrics that will make your pipeline fast and your apps better; 2016. https://www.slideshare.net/CAinc/tech-talk-the-continuous-delivery-metrics-that-will-make-your-pipeline-fast-and-your-apps-better. Accessed 2018.
19. Hamunen J. Challenges in adopting a devops approach to software development and operations [Master's thesis]. Aalto University School of Business. Runeberginkatu 14-16, 00100 Helsinki, Finland: 2016.
20. Silberschatz A, Galvin PB, Gagne G. *Operating System Concepts*. 9th ed. New York, NY: Wiley Publishing; 2012.
21. Kubernetes 2018. https://v1-9.docs.kubernetes.io. Accessed 2018.
22. OpenShift 2018. https://docs.okd.io/3.9/welcome/index.html. Accessed 2018.
23. Conductor 2019. https://netflix.github.io/conductor/. Accessed 2019.
24. Etcd 2018. https://coreos.com/etcd. Accessed 2018.
25. Jenkins 2018https://jenkins.io/doc. Accessed 2018.
26. OpenShift Pipeline Plugin 2018. https://github.com/jenkinsci/openshift-pipeline-plugin. Accessed 2018.
27. Apache JMeter 2018. https://jmeter.apache.org/usermanual/index.html. Accessed 2018.
28. Shadija D, Rezai M, Hill R. *Towards an Understanding of Microservices*. Huddersfield, UK: IEEE; 2017:1-6.
29. Giallorenzo S, Lanesi I. *Choreographies for Microservices*. Odense, Denmark: *University of Bologna/INRIA*; 2017.
30. Hunter T II. *Advanced Microservices: A Hands-on Approach to Microservice Infrastructure and Toolingch*. 1st ed. Berkeley, CA: Apress; 2017:73-87.
31. Furno A, Zimeo E. Self-scaling cooperative discovery of service compositions in unstructured P2P networks. *J Parall Distrib Comput*. 2014;74(10):2994-3025. https://doi.org/10.1016/j.jpdc.2014.06.006.
32. Kalantari B, Schiper A. *Addressing the ZooKeeper Synchronization Inefficiency*. Berlin, Heidelberg: Springer; 2013:434-438.
33. Consul 2018. https://www.consul.io. Accessed 2018.
34. Serf Documentation 2019. https://www.serf.io/docs/index.html. Accessed 2019.
35. Ongaro D, Ousterhout J. In Search of an Understandable Consensus Algorithm. *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. Philadelphia, PA: USENIX Association; 2014:305-319.
36. ZooKeeper 3.5 Documentation 2019. https://zookeeper.apache.org/doc/r3.5.5/. Accessed 2019.
37. Hunt P, Konar M, Junqueira FP, Reed B. ZooKeeper: Wait-free Coordination for Internet-Scale Systems. *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*. Vol 8. 9. Boston, MA: USENIX Association; 2010.
38. Stubbs J, Moreira W, Dooley R. Distributed Systems of Microservices Using Docker and Serfnode. *2015 7th International Workshop on Science Gateways*. Budapest, Hungary: IEEE; 2015:34-39.
39. Balalaie A, Heydarnoori A, Jamshidi P, Tamburri DA, Lynn T. Microservices migration patterns. *Softw Pract Exper*. 2018;48(11):2019-2042.
40. Montesi F, Weber J. Circuit Breakers, Discovery, and API Gateways in Microservices; 2016. arXiv preprint arXiv:1609.05830.
41. Microsoft Azure 2019. https://azure.microsoft.com/en-us/. Accessed 2019.
42. Furber SB, Lester DR, Plana LA, et al. Overview of the SpiNNaker system architecture. *IEEE Transactions on Computers*. 2013;62(12):2454-2467. https://doi.org/10.1109/TC.2012.142.
43. Jenkins X. 2019. https://github.com/jenkins-x. Accessed 2019.
44. Guzmán DPC, Gorostiaga F, Sánchez C. Pipekit: A Deployment Tool with Advanced Scheduling and Inter-Service Communication for Multi-Tier Applications. *2018 IEEE International Conference on Web Services (ICWS)*. San Francisco, CA: IEEE; 2018:379-382.
45. Burns B, Oppenheimer D. Design Patterns for Container-Based Distributed Systems. *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'16 Denver, CO: USENIX Association; 2016.108–113.

How to cite this article: De Iasio A, Zimeo E. A framework for microservices synchronization. *Softw: Pract Exper*. 2021;51:25–45. https://doi.org/10.1002/spe.2877