Computing (2023) 105:1933–1956
https://doi.org/10.1007/s00607-022-01139-5

SPECIAL ISSUE ARTICLE

An architectural style for scalable choreography-based microservice-oriented distributed systems

Gianluca Filippone[1] · Claudio Pompilio[1] · Marco Autili[1] · Massimo Tivoli[1]

Received: 30 June 2022 / Accepted: 25 November 2022 / Published online: 10 December 2022

Abstract

Service choreographies are a versatile approach for building service-based distributed systems. Many approaches can be found in the literature tackling different aspects of service choreographies, such as choreography realizability and conformance checking, distributed coordination, formal choreographic languages, and scalability. As of today, choreography scalability has not been specifically addressed through approaches that also consider coordination issues while still decoupling these two related aspects. Scalability is one of the most important properties to be considered when building distributed systems. It enhances the user-perceived performances and influences the overall dependability of the system. In particular, load scalability allows distributed service-oriented systems to effectively handle varying loads without suffering performance degradation. In this direction, microservice-based systems are able to scale thanks to the possibility of replicating those microservices exposed to growing loads, distributing their workload among different instances. By leveraging on our experience in coordinating service choreographies, in this paper, we propose a layered architectural style that allows to realize scalable microservice-oriented choreographies. The architecture integrates a fully-distributed coordination layer capable of ensuring the correct interactions and a load-balancing layer that allows to balance of coordinated requests. We discuss the properties of the proposed architectural style and evaluate its benefits on user-perceived performances.

✉ Gianluca Filippone
gianluca.filippone@graduate.univaq.it

Claudio Pompilio
claudio.pompilio@univaq.it

Marco Autili
marco.autili@univaq.it

Massimo Tivoli
massimo.tivoli@univaq.it

1 Department of Information Engineering, Computer Science and Mathematics, University of L'Aquila, Via Vetoio, 67100 L'Aquila, Italy

Keywords Distributed systems · Microservices · Scalability · Load balancing

1 Introduction

During the last two decades, service choreography has been receiving considerable attention from the research community as a versatile approach for building service-based distributed systems. In business computing, service choreography is a form of service composition in which the interaction protocol among software services is defined from a global perspective [1]. By embracing the main characteristic of distributed systems, the idea underlying the notion of service choreography can be summarised as follows: *dancers dance following a global scenario without a single point of control.*

Many approaches, from theoretical to more practical, have been proposed in the literature to tackle different challenging aspects concerning service choreographies, spanning from choreography realizability checking to conformance checking [2–5], from formal choreographic languages [6, 7] to distributed coordination synthesis [8, 9]. Moreover, several approaches dealt with scalability [10–14], but none of them specifically address the scalability of choreography-based service-oriented systems together with, yet by fully decoupling, coordination issues as we propose in this paper.

Scalability is one of the most important dimensions to be considered in distributed systems as a desired property that enhances user-perceived performances, hence influencing the overall dependability of the system [15]. As a specific dimension of scalability, *load scalability* is the ability of a system to accommodate increasing loads without suffering performance degradation [16]. A distributed system is said to be scalable when it is able to provide satisfiable performances also under heavy loads, being capable of adapting its potential through the replication of components and/or the addition of computational resources [17, 18].

Microservices are a widely-used architectural style that allows to realize flexible and scalable distributed systems. By inheriting the principles of Software Oriented Architectures (SOA), Microservices allow for designing applications as a collection of small, distributed, and loosely coupled software services [19]. Thanks to their isolation and autonomy, microservices can be independently developed, deployed and scaled [20–22]. The scalability of such systems leverages the small size and isolation of microservices, which enables the replication of those that are more exposed to growing loads in terms of the number of requests and may represent a bottleneck for system performances. Thus, the workload of each microservice can be distributed among a set of microservice instances running in different servers/hosts/containers [21].

As for "classical" service-oriented systems, also microservice-based systems can be conveniently realized through decentralized composition approaches, like choreographies, that permit to enhance loose coupling, independence, and flexibility.

In light of the considerations above, in this paper, we leverage our previous work on service choreographies [8, 9] and propose a layered architectural style to realize scalable choreography-based microservice-oriented systems. The architectural style is composed of two main layers: coordination and load-balancing layer. The coordination layer is needed to ensure that microservices behave as prescribed by the choreography specification and avoid undesired interactions. These interactions do not belong to the set of interactions allowed by the choreography and can happen when the microservices collaborate in an uncontrolled way, possibly leading to failures [23]. The load-balancing layer distributes the workload by properly routing the incoming requests among the available microservice instances. This allows to optimize the resource usage, avoid that a single microservice instance is overloaded by incoming requests, avoid bottlenecks, and hence maximize the system performances [21, 24]. The layered nature of the architectural style implies that the interactions are balanced after being coordinated, hence avoiding the routing of undesired interactions to microservices instances.

The main contribution of this paper can be summarized as follows:

1. enhancement of our previous approach for the coordination of choreographies to comply with the microservices style;
2. combination of load-balancing with coordination mechanism to realize scalable choreography-based microservices system;
3. discussion of the architectural style with respect to several architectural properties;
4. experimental evaluation of the benefits brought by the architectural style on user-perceived performances.

The paper is structured as follows: Sect. 2 overviews the background of our work. Section 3 presents the online ticketing case study. Section 4 presents the proposed architectural style. Section 5 evaluates the architectural style with respect to the several design alternatives and reports on the experiment conducted on the online ticketing case study. Section 6 reports some related works, while Sect. 7 concludes the paper by also describing some further research directions.

2 Background

This section provides background notions of our work. In particular, Sect. 2.1 recalls our approach to the synthesis and coordination of service choreographies, whereas Sect. 2.2 overviews the approaches to load balancing.

2.1 Automated choreography synthesis and enforcement

The approach for the coordination of service choreographies that is leveraged in this paper relies on the automated generation of a set of software entities called Coordination Delegates (CDs) [8, 9, 25]. They are properly interposed among participant services and proxify their interactions by coordinating their message exchange with a distributed coordination algorithm. CDs are generated and associated to *consumer* or *prosumer*[1] services when they can be involved in interactions that require coordination. They decouple the business logic of the system from the coordination logic and ensure that the flow of the messages exchanged by participants follows the choreography specification. Figure 1 shows the architectural view of a sample system in

[1] A prosumer is a service that is both a provider and a consumer.

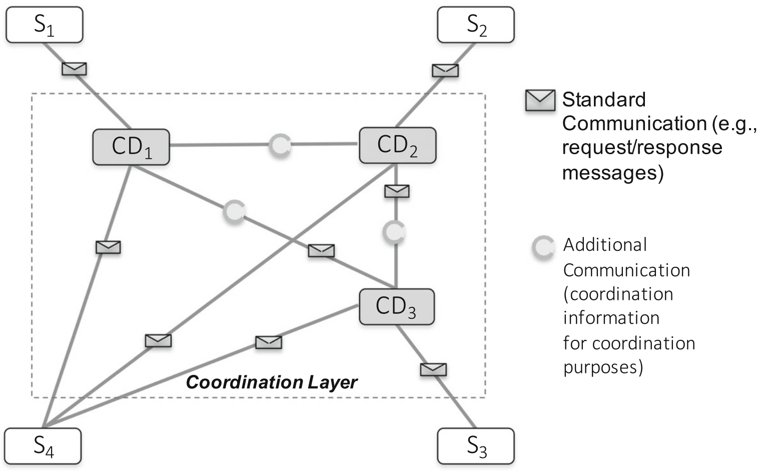

S1
S2
S3
S4
CD1
CD2
CD3
Coordination Layer
Standard Communication (e.g., request/response messages)
Additional Communication (coordination information for coordination purposes)

Fig. 1 Coordination Layer architecture

which four participant services (S_1, S_2, S_3, S_4) are composed together. Their communication is proxified by a *Coordination Layer* composed of three CDs (CD_1, CD_2, CD_3), which coordinate the interaction between services and exchange coordination messages in order to keep track of the global choreography state. When one of the participants sends a business message to another, its associated CD gets the message and waits for the correct choreography state before forwarding it to the receiving participant. This ensures that (i) the messages are exchanged according to the control flow prescribed by the choreography specification and that (ii) no undesired interactions are performed (e.g., for a given service, a certain message has to be received before that another message is sent). Each CD coordinates many choreography executions through a correlation mechanism that is realized by adding a choreography ID in the messages sent by the *consumer* and *prosumer* services. The choreography ID is unique for each execution of the choreography and it is shared among all the CDs, consumer, and prosumer services. It is required to let CDs check the current state of a given choreography execution and enforce the correct sequence of interactions.

Coordination Delegates are generated through an automated synthesis process which takes as input the set of participant services and the BPMN2 model of the choreography, which describes the sequence of message exchanges between the participant services. First, a synthesis algorithm performs a model-to-model transformation in order to obtain the participant models of the choreography. Participant models describe the interaction protocol of each participant (i.e., the sequence of messages that it exchanges), as a partial view of the choreography which considers only the tasks in which each participant (i.e., a service) is involved. Then, a model-to-code transformation carried out from each participant model generates the set of CDs needed to coordinate the whole choreography.

The *Coordination Layer* has been experimentally evaluated concerning the system performances, without detecting issues to consider it a bottleneck. On the other hand, the overall scalability of the system resulted to be weak. In fact, with high demand rates, the execution times of the involved services significantly grow, causing degradation of user-perceived performances [26, 27]. For these reasons, it is needed that, beyond the coordination layer, a load balancing layer is added to the system architecture to support the service scalability, allowing the replication of service instances and supporting the realization of systems in a microservice style. Moreover, also the coordination layer needed to be updated since also Coordination Delegates need to comply with the microservice style, thus supporting the replication of the instances of the coordinated microservices and further improving the coordination layer scalability.

2.2 Load balancing approaches

Load balancing is a fundamental element for scalability in microservice architectures [17, 28]. It allows the distribution of the workloads among multiple microservice instances, running in different physically-distributed containers, to optimize the resource usage. Load balancing avoids that a single microservice instance results overloaded by incoming requests, hence maximizing the system performances. The state of the art for microservice load balancing distinguishes between two approaches: server-side load balancing and client-side load balancing [29, 30].

Server-side load balancing is a centralized approach for distributing requests among microservice instances. Here, a load balancer acts as a proxy that receives requests from clients or consumer microservices and forwards them to target microservices, distributing the workload. In a fully-centralized setting, a central load balancer manages the workload of the instances of all microservice types, having all the traffic passing through it. A more decentralized setting considers having a load balancer per microservice type instead of a central one. Each load balancer proxifies the requests directed to a specific microservice type and manages the workload of its instances.

Client-side load balancing is a fully distributed approach, in which each client or each instance of a prosumer microservice has its local load balancer. Each local load balancer handles the requests coming from the client/prosumer microservice and routes them to the target microservice instance balancing the target's instances workload.

Server-side and client-side approaches have different pros and cons, with many factors influencing their effectiveness and their fitness to the system requirements. Moreover, they require different architectures for supporting load balancing. However, both approaches may be needed for properly balancing the instances of different microservice types in the same system. For this reason, an architectural style able to support both approaches, or even a hybrid approach made by composing the two [24], is desirable.

3 Case study

In this section, we present a case study that allows us to instantiate the challenges described in the previous sections through an example system related to an online ticketing platform. The system has been inspired by well-known applications such as Train Ticket System[2] and Sockshop.[3] It is designed with the specific purpose of realizing a motivating scenario whose characteristics allow to emphasize the needs for both coordination and load balancing and stress the capabilities of the architectural style. The system taken as an example consists of an application that allows customers to browse, select, and buy tickets for sports events or shows. When a customer selects the tickets, the system checks if the customer is allowed to buy tickets, and if so, it will add the selected tickets to her cart and will reserve them for a short period of time until it provides the requested checkout information. If the user proceeds with the checkout within 10 minutes from the ticket selection, the payment is processed and the selected tickets are issued, printed, and prepared for shipping. Otherwise, the temporary reservation is released and the user cart is emptied.

The system is realized as a choreography resulting from the collaboration of several microservices:

- *Event catalog*: it owns the information about the events whose tickets are sold on the platform;
- *Ticket portal manager*: it manages the ticket selling process by interacting with the customer;
- *Blacklist service*: it owns the list of blacklisted people from events (e.g., fans that are not allowed to enter stadiums, etc.);
- *Reservation service*: it owns the information about the tickets that have been sold and the ones that are temporarily reserved because selected by customers while the checkout is performed;
- *Cart*: it manages the user's online cart;
- *Checkout manager*: it manages the checkout process, by receiving the customer checkout information and arranging the payment for the tickets that the customer has in her online cart;
- *Payment*: it processes the payment;
- *Ticketing manager*: it manages the ticket issuing process;
- *Ticket service*: it emits tickets;
- *Print service*: it manages the printing of the tickets;
- *Shipping*: it manages the shipping of tickets to customers.

Figure 2 portrays the BPMN2 choreography diagram of the system. The choreography diagram specifies the message exchanges among the participants (i.e., microservices) from a global point of view. The choreography resulting from the collaboration among participants is modeled as a set of tasks that represent their interactions. A task is an atomic activity that describes an exchange of messages between an initiating participant (which sends the message) and a receiving participant (which receives the message and, optionally, replies with a response message). Graphically, a task is represented by a rounded-corner box labeled with the name of participant services and the name of the task. The initiating participant's name is contained in a white box. Tasks are connected by an arrow, which represents a sequence flow. Parallel paths of execution are depicted by rhombuses marked with "+" which fork the

[2] https://github.com/FudanSELab/train-ticket.

[3] https://microservices-demo.github.io.

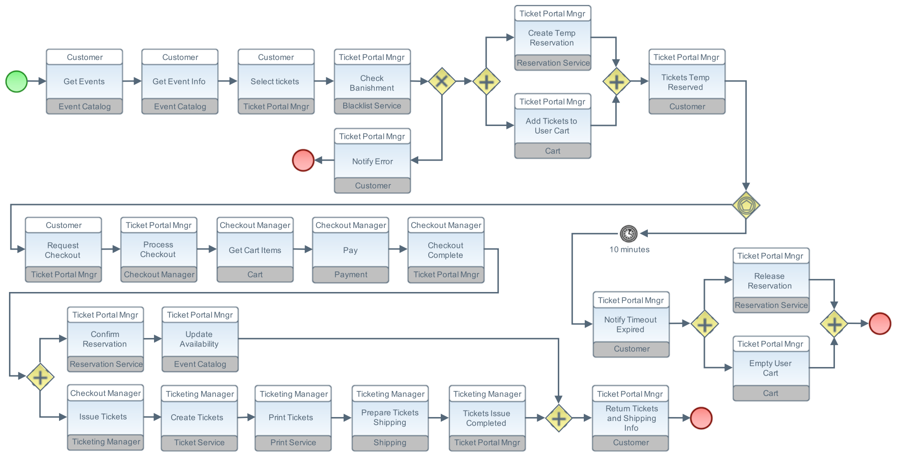


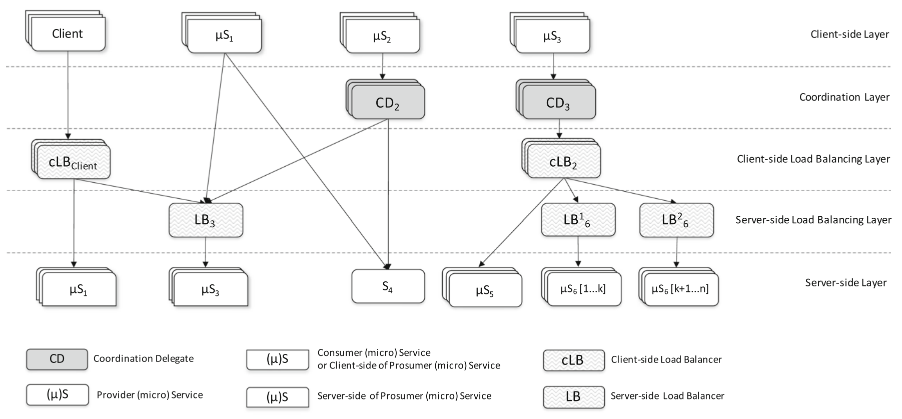
Fig. 2 Online ticketing system choreography

flow in two or more concurrent flows or merge them into a unique flow. Alternatives paths of execution based on conditions are depicted with rhombuses marked with "x" which create exclusive paths within the choreography or merge them into a single path. Moreover, alternative paths can be based on events that occur, such as a message reception (i.e., a task is executed) or a timer event (i.e., a delay mechanism based on a specific time-date) represented with a clock marker.

Note that, besides the microservices listed above, we modeled an extra participant, *Customer*, representing the actor of the system that interacts with the application by sending messages and receiving back replies through an application (e.g., a web-based application, mobile app, etc.).

The realization of this system poses a series of challenges that have to be addressed. First, it has to be considered that, since there are many customers attempting to buy tickets, there are race conditions to be dealt with. In fact, when tickets are selected by a customer, they are temporarily reserved in order to let her complete the checkout process. For this reason, the checkout process needs to be performed only after that the tickets have been reserved: if this does not happen, some other customer may reserve the same tickets and proceed with the checkout at the same time. This may happen if a customer attempts to perform the checkout after that the 10-min timeout has expired. These situations may lead to errors and conflicts in the tickets assignment. For this reason, they clearly represent undesired interactions that require proper coordination in order to be avoided.

Moreover, the system has to be able to scale in order to guarantee responsiveness, availability, and satisfactory user-perceived performances. Stress to system performances happens when the platform opens the sales for an important event with many customers attempting to buy the tickets right on the opening. In these situations, the number of requests that the system is required to handle explodes. The performance of the system may get significantly worse for both customers attempting to buy tickets for the new event and users that are on the platform for other reasons. Hence, the system



Client
μS1
μS2
μS3
Client-side Layer
CD2
CD3
Coordination Layer
cLBClient
cLB2
Client-side Load Balancing Layer
LB3
LB16
LB26
Server-side Load Balancing Layer
μS1
μS3
S4
μS5
μS6 [1...k]
μS6 [k+1...n]
Server-side Layer
CD Coordination Delegate
(μ)S Consumer (micro) Service or Client-side of Prosumer (micro) Service
cLB Client-side Load Balancer
(μ)S Provider (micro) Service
(μ)S Server-side of Prosumer (micro) Service
LB Server-side Load Balancer

Fig. 3 Architectural style

can result unresponsive or unavailable. In addition, different service-level agreements (SLAs) can be applied to provide some users with high performance guarantees. This means that the interactions of certain kinds of users might be prioritized. For these reasons, a load balancing system is needed to properly distribute the requests among the available microservice instances, so as to keep the system available, responsive, compliant to different SLAs, and to avoid the degradation of user-perceived performances.

4 Architectural style

The architectural style we propose in this paper, shown in Fig. 3, is aimed at supporting the scalable composition of microservices. To this extent, the contribution is twofold: (i) the coordination layer is enhanced with respect to the architecture in Fig. 1, and (ii) a load balancing layer is introduced.

4.1 Architectural layers

The architectural style is organized in several layers:

- *client-side layer:* comprises pure client applications as well as the client-side of prosumer services;
- *coordination layer:* contains the coordination logic;
- *client-side load balancing layer:* consists of the client-side load balancers;
- *server-side load balancing layer:* constituted by server-side load balancers;
- *server-side layer:* comprises pure provider services as well as the server-side of prosumer services.

The coordination layer enhancement is driven by the microservice style [19–22]. In fact, CDs now interact with many instances of fine-grained microservices and hence they are replicated as multiple instances employing lightweight communication mechanisms. However, CDs still interact only with the other architecture components, while they are agnostic of the number of their replicated instances. Moreover, the association between CD instances and microservices instances prevents the possibility of the coordination layer to act as a bottleneck and to represent a single point of failure. The multi-instance nature of both CDs and microservices requires load balancing mechanisms to support scalability by distributing the workload across microservices instances. Thus, a load balancing layer is added between the coordination layer and the server-side layer.

The architectural style is flexible with respect to the coordination and the balancing of interactions. In fact, the interactions are coordinated and/or balanced only when needed. Moreover, the two load balancer layers can be combined by leading to a hybrid approach for load balancing [24]. As a result, the architectural style enables the following architectural configurations:

1. Interactions neither coordinated nor balanced.
2. Interactions not coordinated but client-side balanced.
3. Interactions not coordinated but server-side balanced.
4. Interactions not coordinated but hybrid balanced.
5. Interactions coordinated but not balanced.
6. Interactions coordinated and client-side balanced.
7. Interactions coordinated and server-side balanced.
8. Interactions coordinated and hybrid balanced.

These configurations can be divided into two main categories: coordinated and not coordinated interactions. As explained in Sect. 2.1, the coordination of interactions, when needed, enforces the control flow specified in the choreography and avoids undesired interactions. Within each category, we can further identify three sub-categories related to the load balancing approaches applied: client-side, server-side, and hybrid load-balancing. As anticipated in Sect. 2.2, the load balancing approaches have several pros and cons, for example, the server-side approach provides security benefits but represents a single point of failure and a bottleneck. On the contrary, the client-side approach does not represent a bottleneck or a single point of failure but does not address security concerns. The hybrid approach allows to provide different resources availability, costs, or SLAs through dedicated server-side load balancers, but at the same time, it impacts the latency due to the proxy extra hops represented by the server-side load balancers. The flexible nature of the architectural style permits to design a system by using the suitable load balancing approach(es) according to the system's needs.

Concerning the relationships between coordination and balancing layers, we highlight that layers are arranged so that CDs coordinate interactions before that the load balancing is performed, if needed. In other words, load balancers only handle already-coordinated interactions. Hence, the coordination does not impact the load balancer layer(s) since it is performed before that a load balancer routes the request toward the target service instance. Also the choreography ID, required for correlating messages for the purpose of coordination (as explained in Sect. 2.1) is completely transparent to the balancing layers. In synthesis, each layer has a specific concern and does not account for the functionalities offered by other layers, being reciprocally agnostic and their functionalities completely decoupled.

4.2 Architectural style at work

Figure 4 shows the architectural style applied to the online ticketing case study described in Sect. 3. The resulting architecture, beyond handling normal traffic conditions, is capable of dealing with the high traffic related to the ticket sale for an important event by providing different SLAs to users. In fact, premium users are guaranteed better performance and prioritized access to the system by using dedicated microservices instances. In this scenario, the most stressed parts of the system are *Event Catalog*, *Reservation*, and *Ticket Service*. Thus, for each microservice, two server-side load balancers are employed to distribute the workload across the related microservices instances: $\mathrm{LB}^1{}_{EventCatalog}$ and $\mathrm{LB}^2{}_{EventCatalog}$ for *Event Catalog*, $\mathrm{LB}^1{}_{Reservation}$ and $\mathrm{LB}^2{}_{Reservation}$ for *Reservation*, $\mathrm{LB}^1{}_{TicketService}$ and $\mathrm{LB}^2{}_{TicketService}$ for *Ticket Service*. Moreover, the CDs corresponding to the participants interacting with these services, i.e., $\mathrm{CD}_{Customer}$, $\mathrm{CD}_{TicketPortal}$ and $\mathrm{CD}_{Ticketing}$, are equipped with client-side load balancers leading to a hybrid load balancing. In this configuration of the hybrid load balancing, a server-side load balancer is in charge of managing only the traffic generated by premium users. This allows to prioritize premium users according to the related SLA. The hybrid approach is also employed for the microservices: *Ticket Portal Manager*, *Checkout*, and *Ticketing*. Differently from the previous case, the related microservices instances are balanced by a server-side load balancer per microservice type connected with the client-side load balancers of the CDs: $\mathrm{CD}_{Customer}$, $\mathrm{CD}_{Checkout}$ and $\mathrm{CD}_{Ticketing}$. This configuration of the hybrid load balancing routes the traffic of premium users to a dedicated subset of the microservices instances. In fact, the involved microservices handle a light workload, hence it is enough to provide only a dedicated subset of instances to comply with the SLA of premium users. The instances of the microservices *Customer*, *Cart* and *Print* are balanced only by the client-side load balancers of the CDs they interact with, i.e., cLB_{Portal}, $\mathrm{cLB}_{Checkout}$ and $\mathrm{cLB}_{Ticketing}$. The hybrid load balancing is not required for *Customer*, *Cart*, and *Print*. Indeed, due to their nature, they manage simpler and fewer requests with respect to the other microservices, and hence their interactions can be properly balanced through client-side load balancers. Finally, the services: *Blacklist Service*, *Payment*, and *Shipping* are represented as single-instances since they are third-party existing services.

5 Evaluation

As anticipated in Sect. 1, the realization of scalable choreography-based microservices system is obtained through the combination of coordination and load-balancing mechanisms. As said in Fig. 2, the coordination layer has been evaluated without resulting in a bottleneck for the system performances. Thus, the evaluation conducted in this paper focuses on the load balancing layers, i.e., client-side and server-side load balancing layer. Therefore, from the several architectural configurations enabled by the proposed

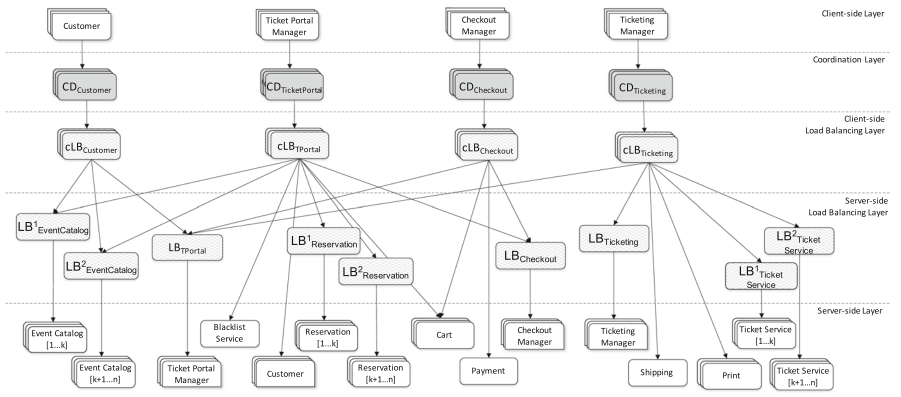


Fig. 4 Online ticketing system architecture

architectural style (Sect. 4), we derived the following design alternatives: *Client-Side Load Balancing*, *Server-Side Load Balancing* and *Hybrid Load Balancing*.
Evaluations goals: the evaluation concerns the following three aspects of the architectural style and answers the related Evaluation-Questions (EQs):

- *Scalability support:*
  EQ1: How is scalability supported by the architectural style?
- *Dependability support:*
  EQ2: How is dependability supported by the architectural style?
- *User-perceived Performances impact:*
  EQ3: What is the impact of the load-balancing layers on the User-perceived Performances?

Evaluations method: the evaluation is carried out both as an *Argumentation* and a *Technical Experiment* [31]. In particular, the *Argumentation* is used to answer *EQ1* by arguing the support of the design alternative with respect the architectural properties: Load balancing scalability (LB Scalability) and Server-side scalability (SS Scalability); whereas *EQ2* is answered by arguing about the architectural properties: User-perceived Performances (UP Performances), Evolvability, Reliability, and Security. Moreover, to quantify the impact of the load-balancing layers on the UP Performances, and hence to answer *EQ3*, a *Technical Experiment* concerning the implementation of the online ticketing system (Sect. 3) has been conducted.

5.1 Argumentation

Table 1 summarizes the results of the argumentation of the design alternatives with respect to the considered architectural properties. In particular, for each design alternative, the table reports if a specific architectural property is supported in a poor (*), moderate (**), or good (***) way.

Concerning User-perceived performances, the client-side load balancer uses a direct connection with the server without introducing proxy extra hops. Moreover, it does not

Table 1 Design alternatives evaluation

Design alternative	UP performances	LB scalability	SS scalability	Evolvability	Reliability	Security
Client-side	***	***	*	*	***	*
Server-side	*	*	**	***	*	***
Hybrid	**	**	***	**	**	***

Good: ***, Moderate: **, Poor: *

represent a bottleneck because it handles the traffic of its local microservice instances in a distributed way [30]. Thus, the client-side load balancer supports the User-perceived performance in a good way. Conversely, the server-side load balancer represents a bottleneck and it introduces an extra hop [32, 33], hence, it supports the User-perceived performance in a poor way. The hybrid load balancing mitigates these disadvantages. In fact, the negative effects of the extra hop and the bottleneck represented by the server-side load balancer are reduced if the system resources and the number of service instances per server-side load balancer are properly determined. Thus, the hybrid load balancing supports user-perceived performance in a moderate way.

Regarding scalability, we consider two dimensions: load balancing scalability (LB Scalability) and server-side scalability (SS Scalability). The former refers to the ability of the load balancer to scale with respect to the number of interactions, whereas the latter involves the ability of the load balancer to support the scaling of the microservice instances. Load balancing scalability is supported in a good way by the client-side load balancer since it can add new load balancing capacity to the system each time a new microservice instance appears. Server-side load balancer supports load balancing scalability in a poor way because it can handle incoming requests by applying vertical scaling up to the availability of the computational resources. The hybrid load balancer allows a flexible form of scaling by adding or removing both service instances and load balancer instances, and hence it supports load balancing scalability in a moderate way.

Server-side scalability is supported in a poor way by the client-side load balancer since it does not manage the number of microservice instances it can use. In fact, a client-side load balancer can't understand whether to add or remove microservice instances, it just distributes its requests among the list of available instances and it can possibly lead to an uneven load distribution [30]. Contrariwise, a server-side load balancer has a continuous awareness of the workload of each microservice instance and hence, according to the traffic, it can easily figure out whenever microservice instances should be added or removed. Moreover, the knowledge of the traffic on the microservice instances allows the server-side load balancer to implement more accurate balancing algorithms. Thus, the server-side load balancer supports server-side scalability in a better way with respect to the client-side. The hybrid load balancer employs both the client-side and server-side load balancer, and hence it allows to distribute requests to suitably partitioned microservice instances complying with resources availability, costs, or service-level agreements (SLAs). Therefore, the hybrid load balancer supports server-side scalability in a good way.

We consider evolvability as the degree to which a component implementation can be changed without negatively impacting other components [34]. A change in a local client-side load balancer implementation may impact the balanced client service resulting in its re-deployment [29]. In fact, client-side microservices may be tightly coupled with the implementation of their client load balancers, (e.g., a microservice and its client load balancer are within the same executable artifact). In such cases, they need to be re-deployed. This introduces a high downtime, and hence a client-side load balancer supports evolvability in a poor way. In a server-side load balancer, the downtime is limited because the change affects only the load balancer [29]. This allows to support evolvability in a good way. The hybrid load balancer supports evolvability in a moderate way because it permits to mitigate the change impact. This can be obtained by applying the change only on the server-side load balancer whenever possible, restricting the implementation of the change on the client-side load balancer only when strictly required.

Reliability from an architectural point of view can be defined as the degree to which an architecture is susceptible to failure at the system level in the presence of partial failures within components, connectors, or data [34]. Client-side load balancer due to its fully distributed nature does not represent a single point of failure, and hence it supports reliability in a good way. Conversely, the server-side load balancer introduces a single point of failure and it supports reliability in a poor way. Hybrid load balancer ease this disadvantage by employing several server-side load balancers. Thus, it supports reliability in a moderate way.

Regarding security, server-side load balancer hides the internal structure of both the application and the network because clients can't contact directly the microservice instances. Thus, server-side load balancer supports security in a good way. On the contrary, client-side load balancer allows clients to directly interact with the microservice instances, and hence it supports security in a poor way. Hybrid load balancer, because of the presence of server-side load balancer, mitigates completely the problems of the client-side load balancer, thus providing good support for security. In fact, in the hybrid setting the client-side load balancer can only interact with the server-side load balancer, hence clients are not aware of the internal network and possible malicious interactions are prevented [24].

5.2 Experimentation

As anticipated, we performed an experiment by implementing and running the system presented in Sect. 3.

This experimentation had the purpose of evaluating the benefits of the presence of a load-balancing layer on the user-perceived performances, in particular in those situations in which there is a high traffic load due to the presence of a large number of users that are accessing the system. Thus, as an indication of the user-perceived performances, we measured the response times for the tasks that involve the *Customer* service (i.e., the user): (i) *Get Events*, (ii) *Get Event Info*, (iii) *Select Tickets* (by measuring the time between the request sent by *Customer* for that task and the reply message received for *Tickets Temp Reserved*), and (iv) *Request Checkout* (by measuring the time between the request sent by *Customer* and the reply message received for *Return Tickets and Shipping Info*). We have run the system by executing the choreography with an increasing number of concurrently interacting users and compared the obtained results for two different scenarios: (i) without any load balancing layer (*unbalanced scenario*), and (ii) with the two load balancing layers by configuring the system as described in Sect. 4.2 (*balanced scenario*). The experimental implementation of the system is publicly available.[4]

Experimentation setting

The goal of the experimentation is to specifically evaluate the impact of the load balancing layers on the user-perceived performances, by comparing the results obtained in the two different settings. Since the architecture only focuses on coordination and load balancing, we avoid considering aspects such as how to distribute, replicate and scale databases, being independent of our architectural style. Also, we do not address the problem of guaranteeing consistency among the distributed and possible replicated databases. Thus, we excluded the persistence from our tests to avoid that results could be biased and influenced by possible bottlenecks in the data access, since it may impact system performance.

In order to perform tests, *Customer* service has been implemented as an application that simulates the presence of a varying number of users that concurrently interact with the system. For each simulated user, it sends the requests to the system according to the execution flow prescribed by the choreography definition: it first sends a message to execute *Get Events* task, then a message for *Get Event Info*, then a message for *Select Tickets*, and at the end a message for *Request Checkout*. The implementation of *Customer* is realized by leveraging the Locust[5] load testing tool. It allows to simulate the behavior of the user by considering also thinking times (randomly selected in a range from 5 to 10 seconds) before sending a message after receiving the response for the previous task. Tasks are executed according to the behavior specified in the choreography, so to simulate the correct sequence of interactions. In this way, we assume that *Customer* is always behaving properly, i.e., it does not attempt to perform undesired interactions as explained in Sect. 3, since the effectiveness of the coordination has been already evaluated in [26, 27] and it is out of the scope of this experimentation.

Data have been collected by the *Customer* service, which locally logs the timestamps of each of the measured interactions listed above, in a way that the collection of data does not interfere with the message exchanges between microservices, CDs, and load balancers. Only when all the instances have been executed, logs are collected and analyzed in order to extract the system response times for each of the operations.

Each of the two scenarios has been tested by running the choreography multiple times with an increasing number of concurrent simulated users, starting from 50 up to 10000, in order to simulate both situations with low traffic and situations with very high traffic.

The experimentation has been performed using eight Virtual Machines (VMs) installed in four distinct Server Machines (SMs). Each SM is equipped with 2CPUs Intel Xeon E5-2650 v3, 2.3 GHz, 64 GM RAM, and 1 Gb/s network. Each VM has

[4] https://github.com/sosygroup/microservices-ticketing-system-prototype.

[5] https://locust.io.

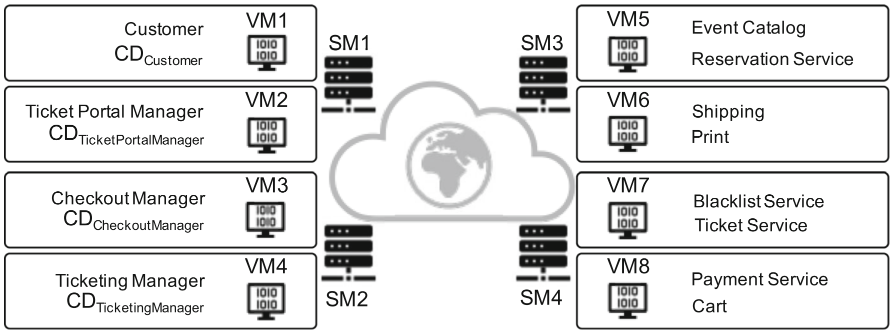


Fig. 5 Deployment setting for the scenario without load balancing

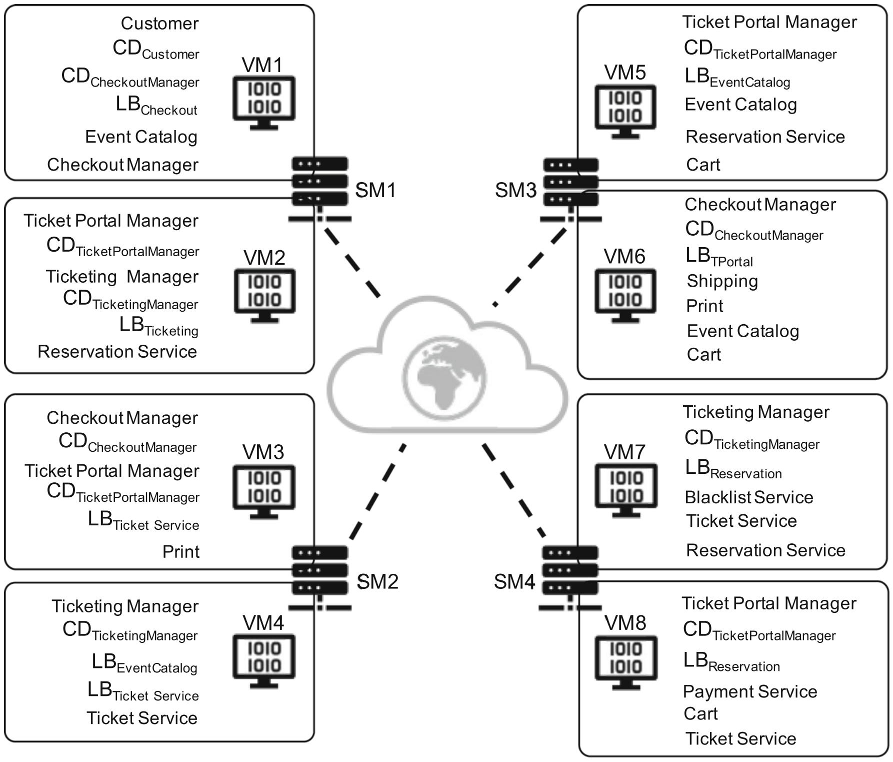


Fig. 6 Deployment setting for the scenario with load balancing layers (CDs are equipped with local client-side load balancer)

4 CPU cores and 4 GB of RAM. The VMs are equipped with Ubuntu Server 22.4 as operating system. Open Stack is the cloud infrastructure provider. Computational resources (CPU cores, RAM, network) of SMs are equally allocated to VMs. Thus, we deployed all the components by following a round-robin approach, except for the CDs, which have been conveniently put (although not mandatory) into the same VM with the consumer or prosumer services they coordinate. In this way, services are

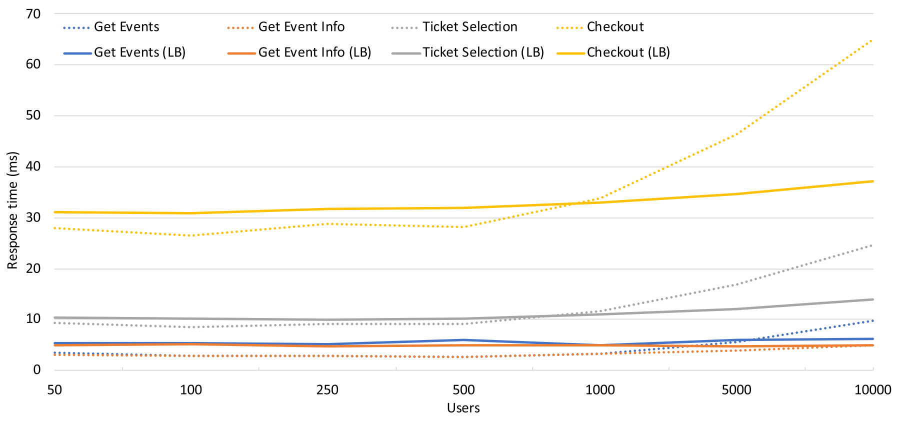


Fig. 7 Experimentation results

uniformly distributed into the VMs to equally distribute the load among them and to avoid resources being consumed in a uniform way. In general, our approach does not constrain the deployment setting, being all the components free to be deployed according to any strategy. The deployment settings related to the two scenarios mentioned above are described in Fig. 5 (unbalanced scenario) and Fig. 6 (balanced scenario).

Experiment results

The chart in Fig. 7 shows the results of the experimental runs. Dotted lines describe the trend of mean response times of the four considered interactions in the unbalanced scenario. Continuous lines, instead, describe the trend in the balanced scenario.

The first thing that leaps into the eye is that in the unbalanced scenario, when the number of concurrent users increases over 1000, the response times raise in a significant way. In particular, the mean response time for the *Checkout* interaction raises from 33.7 ms when there are 1000 concurrent users (medium load) to 65 ms when there are 10,000 concurrent users (very high load), hence doubling the value. The same holds for the interactions *Get Events* (response time raising from 3 to 9 ms) and *Ticket Selection* (raising from 11 to 24 ms). *Get Event Info* showed a response time raising from (3.2 to 4.9 ms). Despite the fact that the increase in response times is sub-linear with respect to the increase in the number of concurrent users (response times double when the number of concurrent users has a ten-times growth), in this experimental setting a degradation of user-perceived performance can be observed. On the contrary, in the balanced scenario, the raise of the mean response times is reduced significantly (from 34.6 to 37 ms for *Checkout* interaction). Thus, the load balancing resulted to be able to effectively reduce the loss of user-perceived performances when the system undergoes high demand rates.

Moreover, we can observe that with very low to medium loads (50–500 concurrent users), the mean response times for all the considered interactions in the balanced scenario are higher with respect to the unbalanced. This is likely due to the presence of server-side load balancers, which require an extra-hop in the message passing between the microservices involved in the interactions. However, the difference between the mean response times in the two configurations is negligible (4.6 ms at most), while the load balanced setting significantly outperforms the unbalanced one in supporting the system's scalability and, hence, avoiding the degradation of user-perceived performances.

6 Related work

The works related to this paper concern approaches for choreography development and scalability, microservice composition, and architecture for load balancing of microservices.

Choreography development and scalability

Scalas et al. [35] present a toolchain that, starting from a choreographic specification encoded in Scribble [36], generates high-level typed APIs for multiparty session programming in Scala. It employs channel-based communication that developers use according to a choreography. We argue that the layering envisioned in our architectural style could be reused in this setting by connecting the channels with the load balancers.

Giallorenzo et al. [37] propose ChIP (Choreographic Integration Process), an integration process employing choreographic programming [38, 39] to generate local connectors that, analogously to our CDs, implement the distributed logic. Thus, our architectural style is amenable to be adopted in this context by equipping connectors with a load balancing layer to make use of it.

Similarly to choreographic programming, multitier programming [40] shares the idea of defining the behaviour of multiple participants in a single compilation unit and then synthesize executable implementations for each participant. Based on this correlation, choreographic programs can be derived from multitier programs, and vice versa [41]. The multiparty system specification could be enriched with the definition of load balancing strategies in order to synthesize the related load balancing artifacts.

In the Internet of Things (IoT) context, an early stage work from Dar et al. [10] exploits the concepts of service orchestration and choreography to address the flexible and adaptive composition of Very Large Scale (VLS) IoT systems. The authors do not propose a specific method to handle the scalability issue, but by combining service orchestration and choreography, they aim at bridging the the gap between physical and cyber world accounting for the scalability and dynamicity issues of IoT. In particular, service orchestration is exploited at the IoT level to integrate sensors and smart devices, while choreography is used to interact with external services.

Furtado et al. [11] present a middleware able to to automatically deploy and execute web services choreographies. It is also capable of monitoring the execution to perform reconfiguration based on SLA constrains to achieve the required QoS. The possible reconfiguration might consist in upscaling or downscaling the resources, increasing or decreasing the nodes executing a service or moving a service to a node with larger/smaller capacity.

Vincent et al. [12] propose a middleware within the CHOReOS project that enable the deployment and enactment of large-scale choreographies in cloud computing environments. It copes with scalability issues by considering QoS aspects both in the selection of resources and in the monitoring of the application.

Barker et al. [13] compare choreography and orchestration for data-intensive computing. In particular, they argue that the increase in the number of services and the size of the data of workflows push scalability to its limits in orchestration techniques. Thus, they demonstrate through several workflow patterns the advantages that choreography brings to data-centric workflows with respect to scalability.

Yoon et al. [14] propose a distributed service choreography framework to resolve semantic conflicting interactions among services. The framework employs safety constraints to avoid conflicting behavior and it reduces the run-time overhead by strategically placing distributed choreography agents and coordinators. The experimental evaluation of the framework shows a negligible overhead in preventing semantic conflicts and a higher scalability than a centralized approach.

Microservice composition approaches
Oberhauser presents Microflows [42], a lightweight and automated approach for the orchestration of semantically-annotated microservices through agent-based clients. Microservices are provided with semantic metadata in support for their automated invocations. They are mapped to nodes and then represented in a graph. Clients act as agents and execute a workflow by planning the shortest path into the graph.

Yahia et al. [43] propose Medley, an event-driven microservice composition platform. It is based on a DSL for producing an orchestration description without the need to focus on the communication issues. The code is compiled into a lower-level code and run on a event-based platform. Medley handles the service adaptation according to service availability and supports horizontal scaling among clusters of nodes.

The work by Monteiro et al. [44] aims to overcome the limitations of the approaches proposed by Oberhauser and Yahia due to the dynamic location of microservices. They propose the use of declarative business processes, which focus on what should be done in order to achieve a business goal rather than all possible alternative flows to be followed. The orchestration of microservices is realized through the Beethoven platform. It is realized through a layered architecture, whose core is the orchestration layer composed of an event bus and an event processor that processes messages, handles tasks, and balances the workload. The orchestration is modeled through a specific DSL.

Gutierrez-Fernandes et al. [45] propose the use of process engines as microservice platform instead of using orchestrators when the business logic of microservices involves complex workflows. Business process languages such as BPMN can be used as a high-level language to define the microservices behavior and their message passing. To this extent, Valderas et al. [46] defined a microservice composition approach as a choreography of BPMN fragments. Their approach considers the developement of a BPMN2 diagram representing the "big picture" of the system, which is then splitted into fragments, each representing the behavior of a microservice. A BPMN engine is used in order to deploy and execute the microservices, while a message bus allows the communication among them.

Sun et al. [47] propose an approach to model variable microservice-based system by using VxBPMN4MS, an extension of BPMN for supporting variability. Microservices invocations are explicitly modeled in the business process and are manipulated in the composition. After the modeling phase, the business process is automatically transformed in a microservice composition framework called AMS implementing the business logic. The composition framework monitors the microservice execution, handling variability and runtime exceptions.

Our approach leverages on a similar technique (BPMN2 diagrams) for the modeling of the microservices choreography. However, while the reported approaches consider the microservices internal behavior, we only focus on their interaction protocols, thus considering (micro)services as black-box entities that have to be externally coordinated. Moreover, none of these approaches considered explicitly dependability aspects since they mainly focused on development and composition issues. In contrast, our layered architecture leverages on the automated choreography synthesis approach in [25] and provides specific support for scalability in microservice composition through the load balancing layer.

Besides the aforementioned Medley and Beethoven, other approaches for the microservice composition considered the use of a specifically-developed programming language. In particular, the Jolie web-service oriented programming language[6] provides support for microservices development by shifting the focus from the computation and frameworks to the communication and dependencies among microservices [48]. By taking inspiration from WS-BPEL, Jolie allows to define the interfaces and the workflow of each microservice, separating the definition of the behavior of a microservice from its deployment.

Architectures for microservices load balancing

Many works in the SOTA about load balancing are specifically aimed at analyzing techniques and algorithms for efficiently performing the service selection process and optimize the load balancing among the system computational resources [49]. These are fundamental aspects since an efficient selection process allows to speed up the communication among microservices, while an optimal balancing strategy allow to guarantee high performances when the system load increases. However, these aspects are out of the scope of our work, since we focused on the architectural aspect rather than on a specific load balancing algorithm.

In [50], the author briefly discusses different load balancing strategies. Beyond the server-side and the client-side strategies, the author portrays an "external" load balancing strategy. Here, clients can communicate directly with server, after they have requested to an external load balancer which instance (i.e., which URI) they have to connect to. In this external approach, there are neither extra ops nor proxified interactions, while the client complexity is smaller with respect to a client-side approach, and the scalability and performances are improved with respect to a server-side approach. However, the external load balancer still represents a single-point of failure, while clients have to still manage some complexity (i.e., the communication with the external load balancer) and they must be trusted.

In [30] Baker Street,[7] a framework for realizing client-side load balancing in microservice systems, is presented. By following the architectural style of client-side load balancing, each client service is provided with an instance of a local load balancer that handles its local traffic. In order to manage the information about availability and location of the microservice instances, each instance has (i) a health checker that reg-

[6] https://www.jolie-lang.org.

[7] http://bakerstreet.io/

ister the microservice to a global discovery service, and (ii) a routing system which owns and updates a local list of the service instances. When a microservice has to send a message, its routing systems proxify the communication by routing the message to one of the available target microservice instances. As argued by the author, while this approach owns all the advantages of a client-side load balancing, it may distribute the load in an uneven way since each load balancer may randomly route the traffic towards the same microservice instances. The use of an hybrid approach, as envisioned in our architecture, would allow to mitigate this issue.

Niu et al. [33] argue that besides the overhead introduced by load balancers that proxify the interaction among microservices, load balancing techniques ignore the competition for shared microservices among different chains of requests that connect a microservice to another. In order to overcome this issue, they present a chained-oriented load balancing algorithm (COLBA), which balances the microservice instances according to the request chains, isolating microservices across them. Their approach allows to steer requests only through microservices that belong to the same chain. They address the challenge by modeling the load balancing as a non-cooperative game and employ a convex optimization technique to obtain an approximated optimal solution to the problem. Similarly, in [51] is pointed out that the load on a microservice instance directly depends on the load distributed on its predecessor instances in the request chain. Considering this, the relationships between microservice instances is modeled as a directed graph, and a QoS-aware load balancing problem is formulated. However, obtaining such kinds of load balancing requires the knowledge of all the request chains across the microservices, besides the fact that the load balancing would require a complete update if some microservice changes and the request chains change as a consequence. In contrast, our load balancing architecture is agnostic with respect to the system's topology, and it is robust with respect to the update, addition or removal of microservices. Moreover, similar advanced balancing techniques can be still implemented as upgrades of the load balancers internal logics.

7 Concluding discussion

In this paper, we proposed an architectural style for the realization of scalable choreography-based microservice-oriented systems. The architectural style is mainly composed of a coordination and a load-balancing layer. The coordination layer is composed of a set of Coordination Delegates that proxify the communication among microservices. They are in charge of coordinating the interactions among participant microservices to enforce the correct choreography execution and avoid undesired interactions. The load balancing layer hosts load balancers, which are responsible for the workload distribution among microservices instances.

The architectural style has been discussed concerning several architectural properties: user-perceived performances, scalability, evolvability, reliability, and security. The flexible nature of the architectural style supports these properties at different levels according to the possible design alternatives.

Moreover, the impact of the architectural style has been experimentally evaluated to quantify its impact on user-perceived performances. The results showed that the load balancing capabilities allow the system to properly scale when it undergoes high loads, effectively preventing the degradation of user-perceived performances.

As future work, we plan to integrate the generation of load balancers into the automated process of choreography synthesis described in [9, 25]. Moreover, we plan to conduct experiments on some real-world case-study to better evaluate the capability of the architectural style in supporting both scalability and system dependability. Finally, we plan to compare the client-side, server-side, and hybrid load balancing approaches with a dedicated experiment to evaluate the differences in the performances of the three approaches enabled by the presented architecture.

References

1. Peltz C (2003) Web services orchestration and choreography. Computer 36(10):46–52. https://doi.org/10.1109/MC.2003.1236471
2. Basu S, Bultan T (2011) Choreography conformance via synchronizability. In: Proceedings of the 20th international conference on world wide web, pp 795–804. https://doi.org/10.1145/1963405.1963516
3. Gössler G, Salaün G (2011) Realizability of choreographies for services interacting asynchronously. In: International workshop on formal aspects of component software. Springer, pp 151–167. https://doi.org/10.1007/978-3-642-35743-5_10
4. Basu S, Bultan T, Ouederni M (2012) Deciding choreography realizability. ACM Sigplan Not 47(1):191–202. https://doi.org/10.1145/2103621.2103680
5. Basu S, Bultan T (2016) Automated choreography repair. In: International conference on fundamental approaches to software engineering. Springer, pp 13–30. https://doi.org/10.1007/978-3-662-49665-7_2
6. Lanese I, Montesi F, Zavattaro G (2015) In: De Nicola R, Hennicker R (eds) The evolution of Jolie. Springer, Cham, pp 506–521. https://doi.org/10.1007/978-3-319-15545-6_29
7. Orlando S, Pasquale VD, Barbanera F, Lanese I, Tuosto E (2021) Corinne, a tool for choreography automata. In: Salaün G, Wijs A (eds) Formal aspects of component software. Springer, Cham, pp 82–92. https://doi.org/10.1007/978-3-030-90636-8_5
8. Autili M, Inverardi P, Tivoli M (2018) Choreography realizability enforcement through the automatic synthesis of distributed coordination delegates. Sci Comput Program 160:3–29. https://doi.org/10.1016/j.scico.2017.10.010
9. Autili M, Di Salle A, Gallo F, Pompilio C, Tivoli M (2019) Aiding the realization of service-oriented distributed systems. In: Proceedings of the 34th ACM/SIGAPP symposium on applied computing. In: SAC '19. Association for Computing Machinery, New York, pp 1701–1710. https://doi.org/10.1145/3297280.3297446
10. Dar K, Taherkordi A, Rouvoy R, Eliassen F (2011) Adaptable service composition for very-large-scale internet of things systems. In: Eyers DM (ed) Proceedings of the 8th middleware doctoral symposium of the 12th ACM/IFIP/USENIX international middleware conference, Lisbon, Portugal, 12 December 2011, pp 2–126. https://doi.org/10.1145/2093190.2093192
11. Furtado T, Francesquini E, Lago N, Kon F (2014) A middleware for reflective web service choreographies on the cloud. In: Costa FM, Andersen A (eds) Proceedings of the 13th workshop on adaptive and reflective middleware, ARM@Middleware 2014, Bordeaux, France, December 8–12, 2014, pp 9–196. https://doi.org/10.1145/2677017.2677026
12. Vincent H, Issarny V, Georgantas N, Francesquini E, Goldman A, Kon F (2010) Choreos: scaling choreographies for the internet of the future. In: Middleware'10 posters and demos track, pp 1–3. https://doi.org/10.1145/1930028.1930036
13. Barker A, Besana P, Robertson D, Weissman JB (2009) The benefits of service choreography for data-intensive computing. In: Rauber T, Rünger G, Jeannot E, Jha S (eds) Proceedings of the 7th international workshop on challenges of large applications in distributed environments, CLADE@HPDC 2009, Garching Near Munich, Germany, June 10, 2009, pp 1–10. https://doi.org/10.1145/1552315.1552317
14. Yoon Y, Ye C, Jacobsen H (2011) A distributed framework for reliable and efficient service choreographies. In: Srinivasan S, Ramamritham K, Kumar A, Ravindra MP, Bertino E, Kumar R (eds)

Proceedings of the 20th international conference on world wide web, WWW 2011, Hyderabad, India, March 28–April 1, 2011, pp 785–794. https://doi.org/10.1145/1963405.1963515
15. Gorbenko A, Kharchenko V, Romanovsky A (2009) Using inherent service redundancy and diversity to ensure web services dependability. Springer, Berlin, Heidelberg, pp 324–341. https://doi.org/10.1007/978-3-642-00867-2_15
16. Bondi AB (2000) Characteristics of scalability and their impact on performance. In: Proceedings of the 2nd international workshop on software and performance, pp 195–203
17. Abbott ML, Fisher MT (2015) The art of scalability: scalable web architecture, processes, and organizations for the modern enterprise. Addison-Wesley Professional, Boston
18. Baresi L, Filgueira Mendonça, D, Garriga M (2017) Empowering low-latency applications through a serverless edge computing architecture. In: European conference on service-oriented and cloud computing. Springer, pp 196–210. https://doi.org/10.1007/978-3-319-67262-5_15
19. Lewis J, Fowler M (2014) Microservices a definition of this new architectural term. https://martinfowler.com/articles/microservices.html. Accessed June 2022
20. Dragoni N, Giallorenzo S, Lafuente AL, Mazzara M, Montesi F, Mustafin R, Safina L (2017) In: Mazzara M, Meyer B (eds) Microservices: yesterday, today, and tomorrow. Springer, Cham, pp 195–216. https://doi.org/10.1007/978-3-319-67425-4_12
21. Dragoni N, Lanese I, Larsen ST, Mazzara M, Mustafin R, Safina L. Microservices: how to make your application scale. https://doi.org/10.1007/978-3-319-74313-4_8
22. Newman S (2015) Building microservices, 1st edn. O'Reilly Media Inc, Sebastopol
23. Autili M, Tivoli M (2014) Distributed enforcement of service choreographies. In: Cámara, J., Proença, J. (eds.) Proceedings 13th International Workshop on Foundations of Coordination Languages and Self-Adaptive Systems, FOCLASA 2014, Rome, Italy, 6th September 2014. EPTCS, vol. 175, pp. 18–35. https://doi.org/10.4204/EPTCS.175.2
24. Autili M, Perucci A, De Lauretis L (2020) In: Bucchiarone, A., Dragoni, N., Dustdar, S., Lago, P., Mazzara, M., Rivera, V., Sadovykh, A. (eds.) A Hybrid Approach to Microservices Load Balancing, pp. 249–269. Springer, Cham. https://doi.org/10.1007/978-3-030-31646-4_10
25. Autili M, Di Salle A, Gallo F, Pompilio C, Tivoli M (2020) Chorevolution: Service choreography in practice. Sci Comput Program 197:102498. https://doi.org/10.1016/j.scico.2020.102498
26. Autili M, Perucci A, Leite L, Tivoli M, Kon F, Di Salle A (2021) Highly collaborative distributed systems: synthesis and enactment at work. Concurr Comput: Pract Exp. https://doi.org/10.1002/cpe.6039
27. Filippone G, Autili M, Tivoli M Synthesis of context-aware business-to-business processes for location-based services through choreographies. J Softw: Evolut Process. https://doi.org/10.1002/smr.2416
28. Balalaie A, Heydarnoori A, Jamshidi P (2016) Migrating to cloud-native architectures using microservices: An experience report. In: Celesti A, Leitner P (eds) Advances in service-oriented and cloud computing. Springer, Cham, pp 201–215. https://doi.org/10.1007/978-3-319-33313-7_15
29. Taibi D, Lenarduzzi, V, Pahl C (2018) Architectural patterns for microservices: a systematic mapping study. In: CLOSER 2018: Proceedings of the 8th international conference on cloud computing and services science; Funchal, Madeira, Portugal, 19–21 March 2018, pp 221–232. https://doi.org/10.5220/0006798302210232
30. Li R (2015) Baker street: avoiding bottlenecks with a client-side load balancer for microservices. https://thenewstack.io/baker-street-avoiding-bottlenecks-with-a-client-side-load-balancer-for-microservices/. Accessed June 2022
31. Konersmann M, Kaplan A, Kühn T, Heinrich R, Koziolek A, Reussner R, Jürjens J, al-Doori M, Boltz N, Ehl M, Fuchs D, Groser K, Hahner S, Keim J, Lohr M, Sağlam T, Schulz S, Töberg, J-P (2022) Evaluation methods and replicability of software architecture research objects. In: 2022 IEEE 19th international conference on software architecture (ICSA), pp 157–168. https://doi.org/10.1109/ICSA53651.2022.00023
32. Kogias M, Iyer R, Bugnion E (2020) Bypassing the load balancer without regrets. In: Proceedings of the 11th ACM symposium on cloud computing. SoCC '20. Association for Computing Machinery, New York, pp 193–207. https://doi.org/10.1145/3419111.3421304
33. Niu Y, Liu F, Li Z (2018) Load balancing across microservices. In: IEEE INFOCOM 2018—IEEE conference on computer communications, pp 198–206. https://doi.org/10.1109/INFOCOM.2018.8486300
34. Fielding RT (2000) Architectural styles and the design of network-based software architectures. PhD Thesis, University of California, Irvine

35. Scalas A, Dardha O, Hu R, Yoshida N (2017) A linear decomposition of multiparty sessions for safe distributed programming. In: Müller P (ed) 31st European conference on object-oriented programming, ECOOP 2017, June 19–23, 2017, Barcelona, Spain. LIPIcs, vol 74, pp 24–12431. https://doi.org/10.4230/LIPIcs.ECOOP.2017.24
36. Honda K, Mukhamedov A, Brown G, Chen T, Yoshida N (2011) Scribbling interactions with a formal foundation. In: Natarajan R, Ojo AK (eds) Distributed computing and internet technology—7th international conference, ICDCIT 2011, Bhubaneshwar, India, February 9–12, 2011. Proceedings. Lecture Notes in Computer Science, vol 6536. Springer, Cham, pp 55–75. https://doi.org/10.1007/978-3-642-19056-8_4
37. Giallorenzo S, Lanese I, Russo D (2018) Chip: A choreographic integration process. In: Panetto H, Debruyne C, Proper HA, Ardagna CA, Roman D, Meersman R (eds) On the move to meaningful internet systems. OTM 2018 conferences—confederated international conferences: CoopIS, C&TC, and ODBASE 2018, Valletta, Malta, October 22–26, 2018, Proceedings, Part II. Lecture Notes in Computer Science, vol 11230. Springer, Cham, pp 22–40. https://doi.org/10.1007/978-3-030-02671-4_2
38. Carbone M, Montesi F (2013) Deadlock-freedom-by-design: multiparty asynchronous global programming. In: Giacobazzi R, Cousot R (eds) The 40th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL '13, Rome, Italy—January 23–25, 2013, pp 263–274. https://doi.org/10.1145/2429069.2429101
39. Preda MD, Gabbrielli M, Giallorenzo S, Lanese I, Mauro J (2017) Dynamic choreographies: theory and implementation. Log Methods Comput Sci. https://doi.org/10.23638/LMCS-13(2:1)2017
40. Weisenburger P, Wirth J, Salvaneschi G (2020) A survey of multitier programming. ACM Comput Surv 53(4):81–18135. https://doi.org/10.1145/3397495
41. Giallorenzo S, Montesi F, Peressotti M, Richter D, Salvaneschi G, Weisenburger P (2021) Multiparty languages: the choreographic and multitier cases (pearl). In: Møller A, Sridharan M (eds) 35th European conference on object-oriented programming, ECOOP 2021, July 11–17, 2021, Aarhus, Denmark (Virtual conference). LIPIcs, vol 194, pp 22–12227. https://doi.org/10.4230/LIPIcs.ECOOP.2021.22
42. Oberhauser R (2016) Microflows: lightweight automated planning and enactment of workflows comprising semantically-annotated microservices. In: 6th international symposium on business modeling and software design (BMSD 2016), vol 1, pp 134–143. https://doi.org/10.5220/0006223001340143
43. Ben Hadj Yahia E, Réveillère L, Bromberg Y-D, Chevalier R, Cadot A (2016) Medley: an event-driven lightweight platform for service composition. In: Bozzon A, Cudre-Maroux P, Pautasso C (eds) Web engineering. Springer, Cham, pp 3–20. https://doi.org/10.1007/978-3-319-38791-8_1
44. Monteiro D, Maia PHM, Rocha LS, Mendonça NC (2020) Building orchestrated microservice systems using declarative business processes. SOCA 14(4):243–268. https://doi.org/10.1007/s11761-020-00300-2
45. Gutiérrez–Fernández AM, Resinas, M, Ruiz–Cortés A (2017) Redefining a process engine as a microservice platform. In: Dumas M, Fantinato M (eds) Business process management workshops. Springer, Cham, pp 252–263. https://doi.org/10.1007/978-3-319-58457-7_19
46. Valderas P, Torres V, Pelechano V (2020) A microservice composition approach based on the choreography of BPMN fragments. Inf Softw Technol 127:106370. https://doi.org/10.1016/j.infsof.2020.106370
47. Sun C-a, Wang J, Liu Z, Han Y (2021) A variability-enabling and model-driven approach to adaptive microservice-based systems. In: 2021 IEEE 45th annual computers, software, and applications conference (COMPSAC), pp 968–973. https://doi.org/10.1109/COMPSAC51774.2021.00130
48. Guidi C, Lanese I, Mazzara M, Montesi F (2017) In: Mazzara M, Meyer B (eds) Microservices: a language-based approach. Springer, Cham, pp 217–225. https://doi.org/10.1007/978-3-319-67425-4_13
49. Afzal S, Ganesh K (2019) Load balancing in cloud computing—a hierarchical taxonomical classification. J Cloud Comput. https://doi.org/10.1186/s13677-019-0146-7
50. Malcom (2018) Load-balancing strategies. https://www.beyondthelines.net/computing/load-balancing-strategies/. Accessed June 2022
51. Yu R, Kilari VT, Xue G, Yang D (2019) Load balancing for interdependent Iot microservices. In: IEEE INFOCOM 2019—IEEE conference on computer communications, pp 298–306. https://doi.org/10.1109/INFOCOM.2019.8737450