Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: 10.1002/cpe.6039

RESEARCH ARTICLE

Highly-collaborative distributed systems: synthesis and enactment at work

Marco Autili[1] | Alexander Perucci*[1] | Leonardo Leite[2] | Massimo Tivoli[1] | Fabio Kon[2] | Amleto Di Salle[1]

[1]Department of Information Engineering Computer Science and Mathematics, University of L'Aquila, L'Aquila, Italy
[2]Department of Computer Science, University of São Paulo, São Paulo, Brazil

Correspondence
*Alexander Perucci, Department of Information Engineering Computer Science and Mathematics, University of L'Aquila, Via Vetoio, 67100 L'Aquila, Italy. Email: alexander.perucci@univaq.it

–
Published version on Concurrency and Computation: Practice and Experience: https://doi.org/10.1002/cpe.6039

Abstract

Service choreographies support a distributed composition approach that is based on the specification of the external interaction of the participant services in terms of flows of message exchanges, given from a global perspective. When developing distributed service-based applications, different services are produced by different teams; at the same time, such choreographies can also interact with third-party services, hence leading to the reuse of black-box services. Enforcing a global coordination logic across the many in-house and third-party services to correctly realize the specified choreography is a non-trivial endeavor. Automatic support is then desirable. In this paper, we present an integrated development and run-time environment for choreography-based systems, which covers all the development activities, including specification, code synthesis, automatic deployment, enactment, and monitoring on the Cloud. We focus on providing a practical solution, i.e., applicable by the community and considering technological standards used in the industry. We report the results of an experiment that we conducted with a use case in the in-store marketing and sales domain. Results confirm confidence in the approach and show that the platform can be applied in practical contexts.

KEYWORDS:
Service Composition; Distributed Coordination; Automated Synthesis; Automated Deployment

1 | INTRODUCTION

The Internet, today, provides many ready-to-use services and promotes the possibility of more-complex value-added services through composition.

Motivation – The evolution of today's Internet is expected to lead to billions of services available soon, radically changing the way modern service-oriented systems will be produced by increasingly reusing and assembling existing services, often black-box [1,2,3]. Thus, the ability to *compose* and *coordinate* software services is of paramount importance. However, in a distributed setting concurrency issues can arise [5], and hence correctly composing and coordinating third-party services is difficult and error-prone. Automated support is then desired.

State of the Art – Choreography [4,5,6] is an emergent composition and coordination style for distributed services. Choreographies describe the global perspective of interactions among the participant services. Choreographies model peer-to-peer communication by defining a multiparty protocol that, when enacted by the cooperating participants, allows reaching the overall choreography goal in a fully distributed way. Choreography does not rely on a central coordinator [7], each service knows precisely when to execute its operations and with whom to interact. Thus, the choreography is a collaborative effort focusing on message exchanging among the participants to reach a common goal [8].

Advancing the State of the Art – So far, choreographies have been solely used for design purposes, simply because there was no technological support for enabling a smooth transition from choreography design to distributed execution and supporting the enforcement of the choreography patterns in automated ways. With "enforcing the choreography" we mean guarantying, from a global perspective, some constraints on interactions

patterns, such as, but not limited to: (i) service A cannot invoke service C before invoking service B; (ii) service A is not allowed to invoke operation X of service B after invoking operation Y of service B; and (iii) services in a group Z can invoke services in a group K, but services in K cannot invoke services in Z. Such examples deal mainly with enforcing the temporality of operations in business flows and the dependency management of groups of services.

In the literature, many formal approaches have been proposed to deal with the problems of checking choreography realizability, analyzing repairability of the choreography specification, verifying conformance, and enforcing realizability [9,10,11,12,13,14,15]. These approaches are based on different interpretations of the choreography interaction semantics, concerning both choreography constructs, and the use of formal notations.

The need for practical approaches to the realization of choreographies was recognized in the OMG's BPMN 2.0[1] standard, which introduces dedicated *Choreography Diagrams*, a practical notation for specifying choreographies that, following the pioneering BPMN process and collaboration diagrams, is amenable to be automatically treated and transformed into actual code. BPMN2 choreography diagrams focus on specifying the message exchanges among the participants from a global point of view. A participant role models the expected behavior (i.e., the expected interaction protocol) that a concrete service should be able to perform to play the considered role.

Problem Space – When a choreography must be employed, for a practical approach, the following problem must be considered: "given a choreography specification and a set of existing services, how to enforce the choreography coordination logic, guarantying the participant services respect the collaboration prescribed by the specification?" To address this problem, possible coordination issues must be solved [5,16,11], which involves non-trivial and error-prone activities. Automatic support is then desirable to enable correct choreography enactment, that is when choreography participants correctly execute their roles by acting independently according to their own interaction behavior [17].

Solution Space – During the last decade, we have been working on choreographies within the context of the EU projects FP7 CHOReOS and its follow-up H2020 CHOReVOLUTION[2]. Differently from most of the approaches previously mentioned, our objective has been to bring the adoption of choreographies to the development practices currently adopted by IT companies. The outcomes of our research and development activities is a comprehensive software platform offering an integrated development and run-time environment for choreography-based systems, which covers all the development activities, including *specification*, *code synthesis*, *automatic deployment and enactment*, as well as *monitoring* on the Cloud [5,18,16,11,19].

Main Contribution – Our contribution focus on providing a practical solution, i.e., applicable by the community, to the problem of enforcing the choreography coordination logic upon a set of already existing services, considering technological standards used in the industry.

In this paper, we focus on the *synthesis processor* and the *enactment engine*, the two main components that offer automatic choreography code generation facilities, and deployment & enactment facilities, respectively. We describe how their interplay enables the systematic transition from choreography design to distributed execution. A use case in the *in-store marketing and sales* domain is used as a reference scenario to show our approach at work, and experimental results are reported. **Results show that:**

i) the overhead due to the execution of the distributed coordination logic is negligible, meaning that the execution of the coordination code automatically produced by the synthesis processor does not affect the choreography "performance" significantly;

ii) the approach is solid, meaning that the coordination code is capable of handling a growing amount of coordination work with no degradation (scalability) as the number of choreography instances running in parallel grows;

iii) the coordination code effectively prevent undesired interactions, meaning that it disallows those interactions that do not belong to the collaboration prescribed by the choreography specification;

iv) the node allocation policies used by the enactment engine can be acted upon to reduce the network delay for exchanging coordination messages.

Paper Structure – The paper is organized as follows. Section 2 overviews the approach and Section 3 introduces the reference scenario, highlights the interaction issues it may have, and how it challenges our approach. Section 4 states the problem and overviews the proposed solution. Sections 5 and 6 describe the choreography synthesis process and the choreography deployment & enactment process. Section 7 describes the experiment that we conducted with the reference scenario. Section 8 reports the results and discusses the lesson learned. Section 9 highlights some peculiarities of our approach and analyzes limitations by also proposing how to cope with them. Section 10 discusses related work, and Section 11 provides a concluding discussion.

[1] www.omg.org/spec/BPMN/2.0
[2] www.chorevolution.eu

2 | APPROACH OVERVIEW

This section gives a high-level overview of the approach by describing the main phases of the approach: *specification*, *code synthesis*, *automatic deployment and enactment*, and *monitoring* on the Cloud.

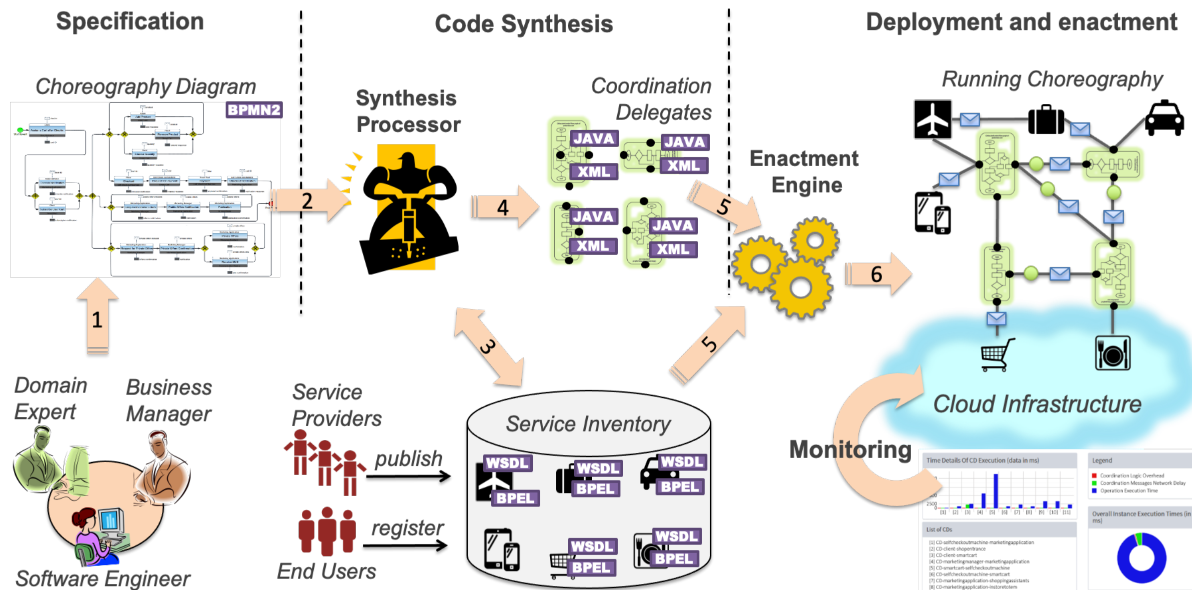

FIGURE 1 Approach overview

As depicted in Figure 1, choreography modelers (such as software engineers, domain experts, and business managers) seat together and cooperate each other to set the desired business goals. For instance, a possible goal might be: assisting visitors in a city from arrival, to staying, to departure. For that purpose, choreography modelers identify the tasks required to achieve the goal and the involved participants. For instance, tasks may concern: reserving a taxi, purchasing digital tickets at the central train station, performing transactions in a shop through services based on near-field communication, etc. Once business tasks have been identified, choreography modelers specify (1) how participants must collaborate as admissible flows of the identified tasks. As detailed in next section, these flows are specified by using BPMN2 choreography diagrams, which are the standard de facto for specifying service choreographies. We also account for the Service Inventory. It contain user registrations and descriptions[3] of software services published by providers that, for instance, have identified business opportunities in the domain of interest. Providers can be local municipalities, airport retailers, bus and/or taxi transportation companies, etc. Out of the choreography specification (2), and by reusing services published into the inventory (3), our synthesis processor automatically generates (4) a set of additional software artefacts, called Coordination Delegates (CDs). As detailed in next sections, CDs implements the distributed coordination logic needed to realize the choreography. Finally, the Enactment Engine resolves the dependencies among services and CDs (5), deploy the achieved choreography-based system on the Cloud and enacts its execution (6). During runtime, the system is continuously monitored by a web application offering a dedicated console. A detailed description of the Enactment Engine and the Cloud infrastructures it supports is given in Section 6.

3 | REFERENCE SCENARIO

Figure 2 shows the BPMN2 choreography diagram of the *In-store Marketing and Sales* use case we used to experiment with the synthesis processor and the enactment engine, and to collect run-time data related to several executions of the corresponding choreography-based system.

BPMN2 Choreography Diagrams (specification) – A BPMN2 task models an atomic activity by specifying an interaction between two participants that exchange one or two messages (request and optionally response) . In a choreography diagram, a task is represented as a rounded-corner box. Each task has a name specified in the middle of the box and involves two participants, one initiating and one receiving, whose roles are also

[3]Description can be WSDL for SOAP services, WADL for REST services together with, optionally, BPEL descriptions

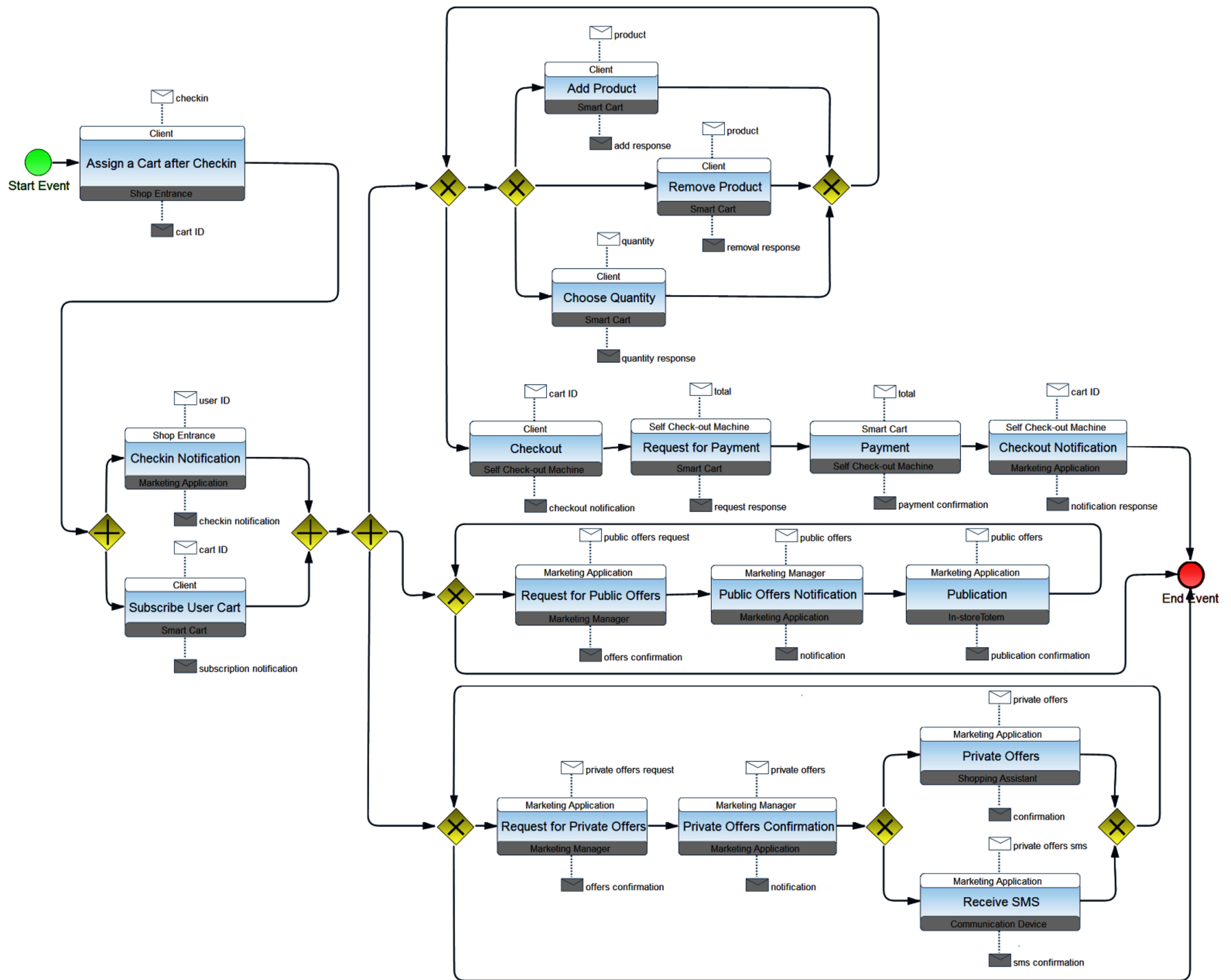

FIGURE 2 In-store Marketing and Sales choreography

specified in the task box as labels of two bands, one white and one grey, respectively. For instance, in the upper-left side of Figure 2, the participant *Client* initiates the task *Assign a Cart after Checkin* by sending the message *checkin* to the receiving participant *Shop Entrance*. Upon task accomplishment, the *Client* can optionally receive the message *cart ID*. Messages are specified by using XML schema[4]. As it will be detailed in next section, participant roles model abstract services that must be then instantiated with concrete services in order to realize the specified choreography.

Scenario Description – The choreography in Figure 2 specifies the allowed interactions of a customer relationship management system while assisting the activity of clients inside a shop and proposing shopping offers and advertisements. Specifically, upon entering the shop, the Client triggers the choreography, and the Shop Entrance service assigns her a cart and notifies the Marketing Application. After subscribing to the cart, the Client can add and remove products from it. In the shop, In-store Totems display public offers to clients. For that purpose, the Marketing Application periodically asks the Marketing Manager for new public offers and pushes them onto the In-store Totems. Similarly, private offers are also proposed to clients through SMS or pushed onto the dedicated Shopping Assistant App. When finished, the Client goes to the Self Check-out Machine and pays for the shopping (in the figure, see the interaction between the Self Check-Out and the Smart Cart services).

BPMN2 Choreography Diagrams (execution) – A choreography does not account for any central control mechanism and there is no mechanism for maintaining any central data. As such, for any action within a single participant, which depends on conditional or assignment expressions, no central way can be assumed for the involved data to be maintained and understood by all (or part of) the participants of the choreography. This means that, due to its distributed nature, the choreography in Figure 2 exhibits a number of complexities that must be dealt with. The specified

[4]www.w3.org/XML/Schema

parallel and alternative flows can in fact give rise to several concurrent execution flows that require to be properly coordinated (in a fully distributed way) in order to realize the collaboration prescribed by the choreography. For that, note the presence of forking branches that must be joined at later execution points (see the rhombuses containing a $+$ with two outgoing flows and their corresponding merging branches with two incoming flows) and the nested conditional branches (see the rhombuses containing a $\times$ with two outgoing flows). The execution semantics surrounding Choreography diagram elements is defined in the OMG's BPMN 2.0 standard specification. Specifically, the BPMN2 employs the theoretical concept of a token that, traversing the sequence flows and passing through the tasks specified by the choreography, aids to define its behaviors. The start event generates the token that must eventually be consumed at an end event. Concerning the execution semantics of parallel ($+$) and exclusive ($\times$) gateways, it is as follows. The parallel gateway (fork and join) is activated if there is at least one token on each incoming sequence flow. This gateway consumes exactly one token from each incoming sequence flow and produces exactly one token at each outgoing sequence flow. If there are excess tokens at an incoming sequence flow, these tokens remain at this sequence flow after the execution of the gateway. In other words, a parallel gateway (fork) spawns new concurrent threads on parallel branches that must be synchronized later at the corresponding parallel gateway (join), which synchronizes multiple concurrent branches, i.e., wait till all concurrent threads complete their execution. Whereas, an exclusive gateway is used to create alternative paths. It has a pass-through semantics for a set of incoming branches (merging behavior). Further on, each activation leads to the activation of exactly one out of the set of outgoing branches (branching behavior). That is each token arriving at any incoming sequence flows activates the gateway and is routed to exactly one of the outgoing sequence flows. In order to determine the outgoing sequence flows that receive the token, conditions expressed as boolean expressions are evaluated in order. The first condition that evaluates to true determines the sequence flow the token is sent to. No more conditions are henceforth evaluated. If and only if none of the conditions evaluates to true, the token is passed on the default sequence flow. In case all conditions evaluate to false and a default flow has not been specified, an exception is thrown.

Undesired Interactions – Moreover, the choreography runs in a distributed setting in which the participants are active entities that execute concurrently and proactively make decisions. This means that the existing services may not synchronize as prescribed by the choreography and, as a result, the global collaboration might exhibit undesired interactions, i.e., those interactions that are not allowed by the specification, but can happen when the services collaborate in an uncontrolled way. Many undesired interactions can be identified in the above scenario. Some of them are described below.

1. After performing the task Subscribe User Cart, Client should not go ahead with its execution if Shop Entrance is still performing Checkin Notification. Client and Shop Entrance are fully distributed and, as such, cannot exploit any central point of control or synchronization to join their respective executions. Thus, Client might go ahead without waiting for joining its execution with Shop Entrance.

2. The Marketing Application should not publish public offers to the In-store Totem if not previously notified by the Marketing Manager.

3. Similarly to public offers, the Marketing Application should not send private offers to the Shopping Assistant nor should it send private offers sms to the Communication Device if not previously requested to the Marketing Manager and confirmed.

4. A Client is allowed to add products to the Smart Cart (see Add Product on the top side of the diagram). After the Payment has been accomplished and before Checking out, a "malicious" client may attempt at adding further products (see the top-most tasks just before the End event), hence avoiding to pay for them.

The ones above are only a few of the undesired interactions that can be easily identified against the choreography in Figure 2. In general, for any choreography and related set of involved participant services, undesired interactions can be identified by considering the following aspects: (i) the order of different sequences of tasks and the timing of the related exchanges of messages; (ii) the various sequences originating in fork gateways that, after being freely executed in parallel, must be suitably synchronized at a later joining gateways; (iii) the exclusivity of the different alternatives flows originating from exclusive branching gateways; (iv) the involved services (as an already existing third-party black-box service being reused) could not be born to perfectly fit the choreography and behavior does not fit the global behavior that emerges from the interaction with the other involved services; (v) dishonest service providers that maliciously modify the behavior of their services. Note that, the last consideration confer a malicious connotation on undesired interactions, as it is the case of the forth interaction listed above.

In general, defining which participant requires coordination, manually calculating its dependencies, implementing the required coordination logic, and automatically deploying and enacting the resulting choreography are non-trivial and error-prone tasks. Therefore, automatic support for the realization of a "correct" coordination logic is needed.

4 | FROM PROBLEM TO SOLUTION SPACE

As introduced in previous section, participant roles in a BPMN2 choreography diagram represent abstract services that, for choreography realization purposes, need to be bound to concrete services, either existing ones or implemented from scratch.

Our approach is reuse-oriented, meaning that it allows to enforce choreography realizability in contexts in which the choreography is not implemented from scratch but it is realized by reusing, as much as possible, third-party services published in a Service Inventory [5]. Thus, concrete services are selected from the inventory to play the roles of the abstract participants modeled by the choreography specification. This calls for exogenous coordination of the selected concrete services since, in general, we cannot access their code and change it whenever needed.

Problem Statement – With this premise, the realizability enforcement problem that we solve can be phrased as follows: *given a choreography specification and a set of existing services (selected as concrete participants), externally coordinate their interaction so to fulfill the global collaboration prescribed by the specification*.

A detailed description of the above stated problem is given in our previous work [5], where we fully formalize the concurrency/coordination issues we are able to solve and the related notion of choreography realizability. For the purposes of this paper, it is sufficient to say that the need to externally coordinate the interaction of the existing services so to fulfill the choreography specification – hence enforcing realizability – arises from the fact that, although the participant services may result perfectly fitting if taken individually, their composition may not fit the specified message exchanges and related flows. This means that the composite behavior of the interacting services can prevent the choreography realization if left uncontrolled or wrongly coordinated, because it may exhibit behaviors that are not modeled by the specification. Thus, solving the problem requires to distribute and enforce, among the participant services, the global coordination logic extracted from the choreography specification.

The synthesis processor permits the realizability enforcement of choreographies via automated generation of additional software entities, called *Coordination Delegates* (CDs). When interposed among the participant services, CDs proxify the services interactions and coordinate them in order to realize the specified choreography in a fully distributed way. The derived CDs are interposed (only if strictly needed) among the participants according to a predefined architectural style (see Figure 3). Diving into a detailed and formal description of how CDs are automatically synthesized, how they are interposed among the services, and what is the distributed coordination algorithm they implement is out of scope. Interested readers can refer to our previous work in [20]. However, to make the paper more self-contained, in the following we briefly discuss some important aspects concerning the automated generation of CDs.

Coordination Delegates Generation – The ability of extracting specific flows of the specified BPMN2 choreography diagram – hereon, referred to as C – represents a basic ingredient to support the automated synthesis of CDs. For instance, in order to deal of several parallel flows to be synchronized in correspondence of a join, the CD synthesis method extracts, out of C, the flows that originate from the interested fork to identify (i) the CDs that must be notified about the reach of the join and, complementary, (ii) the CDs that must be waited for. The former are notified through the broadcast sending of `NOTIFY` coordination messages. The latter are waited for until `WAIT` coordination messages are received from each of them. A further basic ingredient of our synthesis method concerns a distribution step for C. That is, C is distributed into a set of Coordination Models (CMs), one for each CD. We recall that CDs are not always generated for each pair of participants. Indeed, for two given participants, a CD is generated only when there is a dependency relation between them. This means that we do not require to always keep the global state of the choreography to enforce it. This characterizes the distributed nature of our synthesis approach. Each CM represents a local view of C. It is local since it models the coordination logic that has to be enforced on the interaction between two participants, p_i and p_j. This is done by exchanging coordination information (`NOTIFY` and `WAIT` messages) among the CD supervising p_i and the CDs it needs to synchronize with. Thus, the set of all CMs can be considered as a distributed model of C. A CM characterizes the coordination information that is needed to generate the coordination logic that the corresponding CD has to perform, while interacting with the other CDs, in order to realize C. As formalized in [20], CDs automated generation relies on the extracted CMs in order to implement a distributed coordination algorithm. The algorithm inherits the distributed mutual exclusion principle and leverages some foundational notions (such as happened-before relation, partial ordering, and time-stamps) of the algorithm in [21]. More precisely, the enforcement of mutual exclusive access to a single resource in [21] is scaled up to the enforcement of concurrent task flows according to arbitrarily complex choreographies.

In the style of [21], a CD is generated as a *reactive* software entity that, at each iteration of the algorithm, waits for receiving either a request from the service it supervises or `NOTIFY` messages from the other CDs. The most important aspect here is that, upon realizing a join, the involved CDs `NOTIFY` and `WAIT` for each other in order to synchronize their execution. At run time, `NOTIFY` messages (together with a simple notion of *priority*) are used to realize joins as distributed synchronization points. Then, when a join has been realized, and hence all the parallel executions have been synchronized, the CD with highest priority is in charge of `NOTIFY`ing the CDs that, according to the extracted CMs – and, hence, to the specified choreography – are allowed to proceed. In our implementation, the correct co-relation among `WAIT` and `NOTIFY` messages is realized by means of dedicated queues that, for each `WAIT`, buffer the expected `NOTIFY`s. At the beginning, all CDs are waiting for receiving an initiating message to internally set the state(s) from which they can perform an operation. These messages are sent by the Enactment Engine. As detailed in the following, the latter is also in charge of automatically deploying the CDs.

Architectural Style – Figure 3 uses a box/line (i.e., component/connector) notation [22] to show a sample architecture instance conforming to the predefined architectural style. CDs implement a distributed coordination algorithm to perform business-level coordination by intercepting/proxifying the service interaction (*Standard Communication*) and mediating it by exchanging coordination information (*Additional Communication*, i.e., `NOTIFY` and `WAIT` messages) extracted by the synthesis processor out of the choreography specification. Possible *undesired interactions* are thus

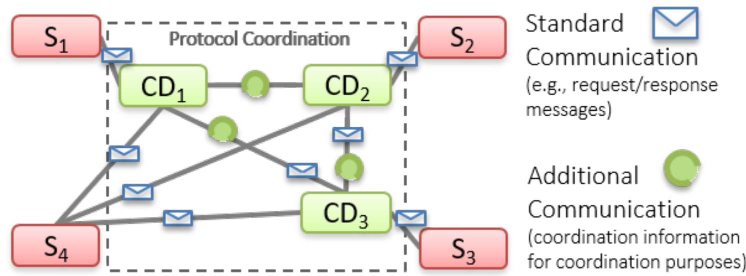

FIGURE 3 Architectural style (sample instance of)

prevented by CDs. We use the term undesired interactions to identify those system interactions that are not permitted by the choreography specification, which can however happen when the involved services collaborate in an uncontrolled manner.

CDs must be deployed in and launched from Internet-accessible servers. A current trend is deploying services in the cloud to increase the degree of automation for deployment. Therefore, the generation of CDs must be followed by their automated deployment. Moreover, CDs must be bound among them and among the participant services, so they can collaborate within the choreography. For deploying CDs, the Enactment Engine selects and/or creates cloud nodes where CDs are automatically deployed, hence binding these CDs among them and the participant services.

5 | SYNTHESIS PROCESSOR

The processor takes as input a Choreography Diagram and the XML Schema of the messages to be exchanged by the participants, and performs the following phases (Figure 4).

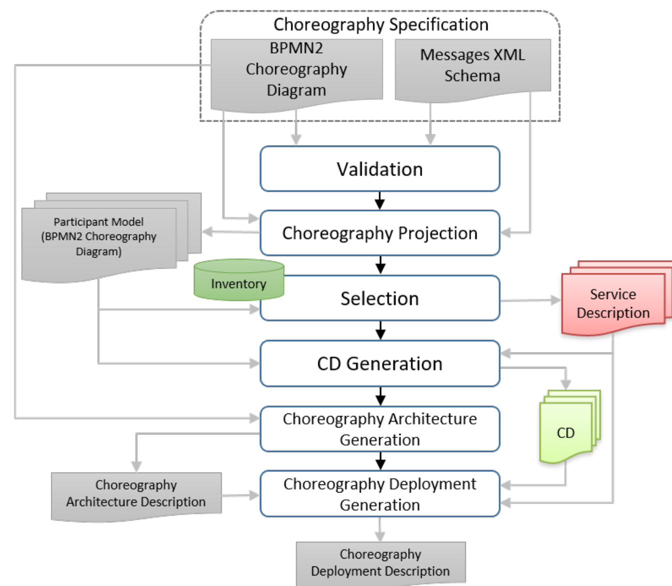

FIGURE 4 Synthesis process

Validation – The goal is to check the validity of the specification against the well-formedness constraints imposed by the OMG's BPMN 2.0 specification, hence ensuring the choreography realizability [10,14,15] and its enforceability [5].

Choreography Projection – A Model-to-Model (M2M) transformation is performed to extract a set of Participant Models out of the choreography specification, one for each participant. A participant model describes the interaction protocol of a single participant. It is a partial view of the

choreography concerning only the flows where the considered participant is involved and, as such, it is still a BPMN2 choreography diagram. For each participant, it represents the expected behavior that a concrete service should be able to perform in order to play the role of the (abstract) participant in the choreographed system to be.

Selection – The CHOReVOLUTION platform also features an Identity Manager (not shown in the figure) based on the Apache Syncope project[5] and it is responsible for managing digital identities of users and services. The Service Inventory in the figure is a sub-component of the Identity Manager which is implemented as a REST API based on the extension mechanism provided by Apache Syncope. It offers registration, search and selection capabilities. Thus, the selection phase exploits the participant models to query the Service Inventory to select concrete services that can play the roles of the participants. For each concrete service, the output is a Service Description model that describes the service in term of its interface and, optionally, its interaction protocol. For instance, the interface could be described in WSDL or Swagger[6] files, and the interaction protocol as a BPEL process or (optionally, for those more familiar with formal notation) a Labeled Transition System (LTS), the latter slightly extended for including those details that enable our automatic synthesis as per the next phases[5,11,20]. However, situations where BPEL or LTS models are not specified for the participating services, choreography developers can manually select concrete services as choreography participants.

CD Generation – The service description models and the participant models are taken as input by our synthesis method for generating the needed CDs. It is worth recalling that the CDs are generated and interposed among the services only if strictly needed, according to the collaboration prescribed by the choreography specification.

Choreography Architecture Generation – Considering the choreography diagram, a Choreography Architecture Description model is automatically generated, together with a graphical representation of it. The architecture resulting from the synthesis of the In-store Marketing and Sales choreography described in Section 4 is shown in Figure 7.

Choreography Deployment Generation – This phase concerns the generation of the Choreography Deployment Description (called `ChorSpec`) out of: (i) the choreography architecture description, (ii) the CDs, and (iii) the descriptions of the selected services. The ChorSpec is an XML file that refers to all the generated artifacts to specify their dependencies. The deployment description is given as input to the enactment engine to deploy and enact the realized choreography-based system into a cloud environment.

6 | ENACTMENT ENGINE

Deployment must be a fully automated process to ensure testability, flexibility, and reliability[23] and to enable the continuous delivery of new versions of the software[24]. Manually deploying distributed systems is a time-consuming and error-prone activity[25], especially for the large-scale ones. To handle such scenario, the Enactment Engine (EE)[19] provides a fully automated deployment process for service compositions.

The EE enables deployment automation by providing a RESTful API that receives the choreography deployment specification and returns information describing the deployment outcome. When receiving the specification, as according to Figure 5, the EE performs the following actions:

Node Selection – selects one or more cloud nodes where each CD will be deployed, creating new nodes if necessary.

Scripts Generation – dynamically creates scripts for environment configuration and CDs launching.

Nodes Update – executes the scripts on the selected nodes, so CDs are installed and launched.

Service Binding – for each participant (original services and CDs), EE provides the addresses of their dependencies (other services and/or CDs), so they can collaborate.

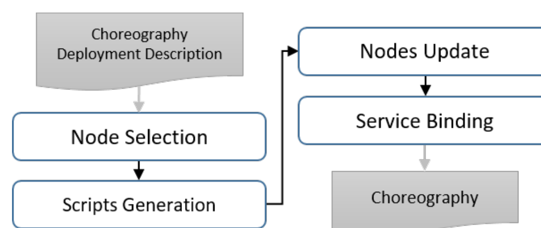

FIGURE 5 Deployment process

[5]https://syncope.apache.org
[6]https://swagger.io/

The deployment specification must provide all the necessary information to deploy the choreography, including, for each service, where the service package can be downloaded from and the type of the package. Out of the box, the EE supports the deployment of WAR and JAR packages. The deployment specification also specifies existing third-party services that are already available on the Internet and must be bound to the choreography dynamically. The deployment specification adheres to the model described in Figure 6 and it is automatically generated by the synthesis processor.

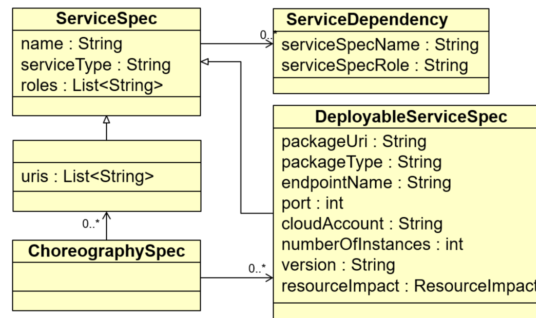

FIGURE 6 Data model defining choreography deployment specification

Each service can consume operations from other services in the composition. Thus, the deployment process must *bind* the services, so they know how to invoke each other. When using the EE, each consumer service must implement an operation called `setInvocationAddress` that receives dependency endpoints. Service dependencies are also declared on the deployment specification, so the EE can properly inject dependencies [26] on dependent services.

Virtualized resources provided by the cloud leverages the automation of the deployment process [27]. The EE relies on Infrastructure as a Service (IaaS) providers to build and manage virtual machines, which enables the creation of a clean environment at each deployment, increasing deployment reproducibility and reliability. Currently, the EE supports both Amazon EC2[7] and OpenStack as infrastructure providers. However, through extensions mechanisms, one can plug a new cloud provider onto the EE without the need to change any of its code. On the other hand, using cloud resources brings additional challenges, since it is necessary to take into account the dynamic nature of the cloud [28], e.g., the service binding must be performed at runtime.

The dynamic nature of the cloud also demands a flexible node allocation policy. It is possible to extend the EE to define dynamic node selection policies based on the available non-functional service requirements. Out-of-the-box, the EE supports the following policies:

- *Always create:* a new VM is created for each deployed service instance;
- *Round robin:* services are deployed in a round robin fashion across a set of predefined VMs;
- *Limited round robin:* initially, new VMs are created for each deployed service until a limit of created VMs is reached. After this, services are deployed in a round robin fashion on the already created VMs.

Enactment engine benefits – The enactment engine by itself enables the choreography enactment, including services deployment and services binding. When used in conjunction with the synthesis processor in our reference scenario (Section 3), the enactment engine automates the deployment of the generated CDs, including binding these CDs to already existing third-party services. Without the enactment engine, software engineers would have to implement a specific deployment process for each choreography produced by the synthesis processor, which would significantly extend the total time required to deliver new choreographies, even considering the use of existing third-party services. The EE presents a good scalability, coping with a larger number of services to be deployed as the number of resources (cloud nodes) grows as well [19].

[7] https://aws.amazon.com/ec2/

7 | EXPERIMENTATION

This section describes the experiment we conducted with the reference scenario described in Section 3. The experiment covers all the phases that follow the choreography specification activity, from code synthesis (Section 7.1), to automatic deployment and enactment (Section 7.2), to choreography execution and monitoring (Section 7.3). Then, in Section 8, we report results and discuss the lesson learned.

We recall that the reference scenario concerns a customer relationship management system which assists the activity of clients inside a shop while proposing customized shopping offers and advertisements. The experiment emulates the simultaneous presence in the shop of a varying number of clients over the course of 24 hours, considerably increasing during peak hours. The experiment purpose is to evaluate (i) the overhead due to presence of CDs, (ii) the scalability of the approach, (iii) the effectiveness of the approach to prevent undesired interactions, and (iv) the deployment performed by the enactment engine.

A replication package is publicly available[8] to support the full and independent replication of the experiment. It includes the artifacts generated by the synthesis processor, the enactment engine, the monitor and extracted raw data.

The experiment was carried out using nine Virtual Machines (`VMs`) installed in three distinct (geographically distributed) Server Machines (`SMs`), three VMs for each server machine (Figure 8). Specifically, the server machine `SM1` is located in the premises of the University of L'Aquila (Italy), `SM2` in the University of São Paulo (Brazil), and `SM3` in the Amazon EC2 cloud infrastructure. `SM1` has 2 CPUs Intel Xeon® E5-2650 v3, 2.3 GHz, 64 GB RAM and 1 Gb/s LAN network. `SM2` has 4 CPUs Quad-Core Intel Core i7®, 4.2 GHz, 64 GB RAM and 1 Gb/s LAN network. `SM3` has an Amazon EC2 A1 instance featured by a custom built AWS Graviton Processor with 64-bit Arm Neoverse cores and by a support for Enhanced Networking with Up to 10 Gbps of Network bandwidth. Each VM has 4 CPU cores, 16 GB RAM, the operating system is Ubuntu Server 14.04 LTS. Open Stack is the cloud infrastructure provider managing VMs from `VM1` to `VM6`.

7.1 | Synthesis

This part of the experiment concerns the synthesis activities in Figure 4. The synthesis processor takes as input the BPMN2 choreography diagram exemplified in Figure 2 and, by following the architectural style described in Figure 3, automatically generates the Choreography Architecture Description model depicted in Figure 7.

The architecture in Figure 7 shows how the synthesized CDs are interposed among the choreography participants. In the figure, the legend associates a number to each participant; the notation $CD_{i.j}$ is used to denote the CD that supervises the participant p_i by proxifying p_j. When

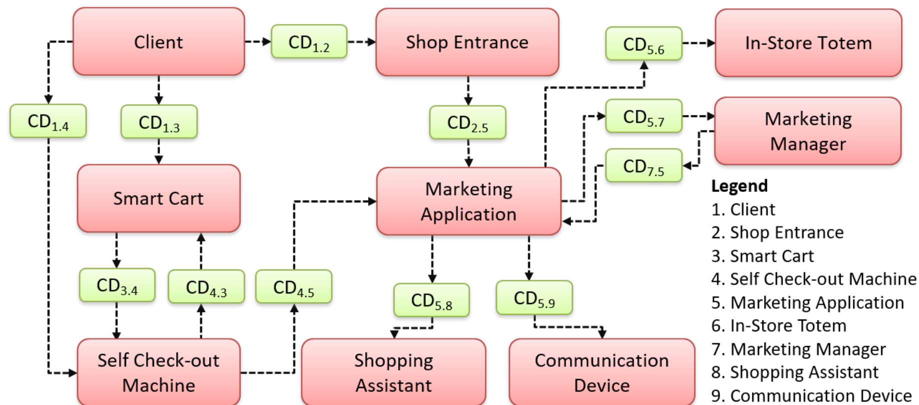

FIGURE 7 In-store Marketing and Sales choreography architecture

interposed among the services to be choreographed, CDs coordinate their interaction in a fully distributed way. As anticipated in Section 4, CDs implement a distributed coordination algorithm according to which each involved $CD_{i.j}$ exchanges coordination information with the other CDs it needs to synchronize with. For instance, with reference to the left-hand side of the choreography in Figure 2, $CD_{1.2}$ supervises `Client` by proxifying `ShopEntrance`, and it needs to synchronize with $CD_{2.5}$ and $CD_{1.3}$ to enforce the execution of the tasks `Checkin Notification` and `Subscribe User Cart` only after the task `Assign a Cart after Checkin` is accomplished. For the sake of presentation, in Figure 7, CDs dependencies are not

[8] https://sesygroup.github.io/choreography-synthesis-enactment

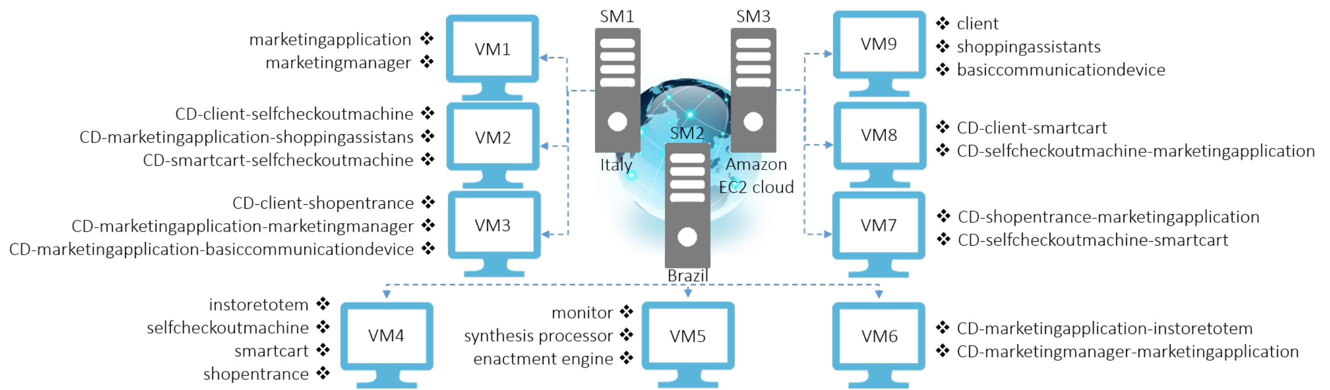

FIGURE 8 Services allocation on the available virtual machines

shown although they are internally stored by following the architectural style in Section 4. Note that CDs are not required for any possible pair of participant services, instead, they are generated only if strictly needed, according to the collaboration prescribed by the choreography specification (i.e., only for those pairs of participants involved in the same task). The approach was conceived to be highly modular and fully distributed. This means that, e.g., in situations in which two participant services are invoked by each other the approach generates two different CDs, e.g., $CD_{4.3}$ and $CD_{3.4}$, and $CD_{5.7}$ and $CD_{7.5}$ (see Figure 7). More on that in Section 9.

Listing 1 shows an excerpt of the `ChorSpec`, as generated by the synthesis processor out of the architecture configuration depicted in Figure 7. The excerpt highlights the dependencies between `Client` and `Shop Entrance`. Briefly, the `ChorSpec` specifies the set of existing services involved in the choreography (see the `legacyServiceSpecs` tag), the set of synthesized CDs (see the `deployableServiceSpecs` tag), and the service-CD and CD-CD dependencies (see `dependencies` tag).

Next section describes how the information contained in the `ChorSpec` is used by the enactment engine to perform choreography deployment and enactment.

```
<choreographySpec>
 ... ... ...
 <legacyServiceSpecs>
  <dependencies>
   <serviceSpecName>CD-client-shopentrance</serviceSpecName>
   <serviceSpecRole>shopentrance</serviceSpecRole>
  </dependencies>
  ... ... ...
   <name>client</name>
   <roles>client</roles>
   <serviceType>SOAP</serviceType>
   <nativeURIs>http://vm9/client/client</nativeURIs>
 </legacyServiceSpecs>
 ... ... ...
 <deployableServiceSpecs>
  <dependencies>
   <serviceSpecName>shopentrance</serviceSpecName>
   <serviceSpecRole>shopentrance</serviceSpecRole>
  </dependencies>
  <dependencies>
   <serviceSpecName>CD-shopentrance-marketingapplication </serviceSpecName>
   <serviceSpecRole>marketingapplication</serviceSpecRole>
  </dependencies>
  <dependencies>
   <serviceSpecName>CD-client-smartcart</serviceSpecName>
   <serviceSpecRole>smartcart</serviceSpecRole>
  </dependencies>
  <name>CD-client-shopentrance</name>
  <roles>shopentrance</roles>
  <serviceType>SOAP</serviceType>
  <endpointName>CD-client-shopentrance</endpointName>
```

```
  <numberOfInstances>1</numberOfInstances>
  <packageType>TOMCAT</packageType>
  <packageUri>http://vm5/cdgenerator/getcd/1422282637249/ CD-client-shopentrance.war</packageUri>
  <port>10002</port>
 </deployableServiceSpecs>
 ... ... ...
 <legacyServiceSpecs>
  <dependencies>
   <serviceSpecName>CD-shopentrance-marketingapplication </serviceSpecName>
    <serviceSpecRole>marketingapplication</serviceSpecRole>
  </dependencies>
  <name>shopentrance</name>
  <roles>shopentrance</roles>
  <serviceType>SOAP</serviceType>
  <nativeURIs>http://vm4/shopentrance/shopentrance </nativeURIs>
 </legacyServiceSpecs>
 ... ... ...
</choreographySpec>
```

Listing 1: Excerpt of the `ChorSpec`

7.2 | Deployment & Enactment

This part of the experiment concerns the enactment engine activities in Figure 5. The enactment engine takes as input the `ChorSpec` and deploys the synthesized CDs. Specifically, the enactment engine reads all the `deployableServiceSpecs` tags to identify the CDs that need to be deployed. Note that the synthesis processor runs on the machine `VM5` (see Figure 8). Therefore, the packages containing the synthesized CDs are stored into that machine and can be downloaded by the EE from the URIs specified in the `packageUri` tags – see `http://vm5/cdgenerator/getcd/1422282637249/CD -client-shopentrance.war` in Listing 1. Note also that `VM5` was used for running the monitor and the EE too.

In the experiment, the services to be choreographed are distributed across the virtual machines `VM1`, `VM4`, and `VM9`. The remaining virtual machines, namely, `VM2`, `VM3`, `VM6`, `VM7`, and `VM8`, are used by the EE to deploy the CDs by applying the round-robin policy (see Section 6). Once the service-CD and CD-CD dependencies are also set, the choreographed system is ready to accept the requests by new clients entering the shop. An important observation here is that, thanks to the usage of three distinct server machines, namely, `SM1`, `SM2`, `SM3`, this deployment setting permitted us to concretely experiment with a real scenario where existing services are offered by different geographically distributed providers.

7.3 | Execution and Monitoring

This part of the experiment concerns the execution of the choreographed system and the collection of runtime data. The goal was to “stress” the system by emulating the simultaneous presence in the shop of a varying number of clients over the course of 24 hours, considerably increasing during peak hours. This permitted us to collect runtime data while the system was required to manage different degrees of parallelism in term of choreography instances being simultaneously executed, and hence different “loads” in terms of coordination work.

For each client entering the shop, a new choreography instance is created. Overall, we executed 4000 instances reaching the highest parallelism degree equal to 257 during the peak hours.

Independently of the parallelism degree, 400 choreography instances were “malicious”, meaning that these instances caused undesired interactions. With reference to Section 3, we recall that a possible undesired interaction can happen when, after the payment of the selected products has been accomplished and before checking out, a client attempts at adding further products (through the “Add Product” operation), hence avoiding to pay for them.

For what concerns the collection of data, we have implemented an ad-hoc monitor capable of obtaining data from the software entities involved in the choreography, and to display these data in a web console for analysis purposes (Figure 9).

Figure 10 shows the events that each software entity logs (see the timestamps 1 to 8). The monitor (which is deployed in the virtual machine `VM5` – see Figure 8) does not collect any data during choreography execution; rather, the logged data are stored locally to each software entity in a way that they are not tainted by the monitoring activity. This means that the monitor does not interfere with the choreography execution and gets the data only after the execution of the choreography is completed. This means that the experiment results reported in the next section come from the analysis of the data logged by each involved software entity, monitoring time excluded.

FIGURE 9 Web choreography console

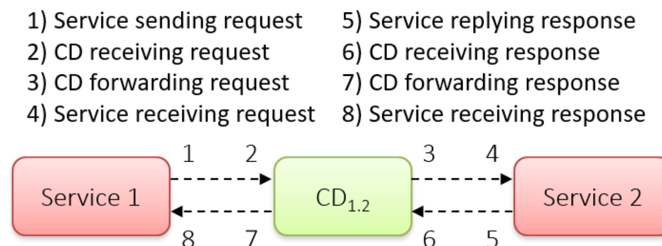

FIGURE 10 Events logged for services and CDs

An important consideration concerning the monitoring activity is that by increasing the number of parallel instances, we are able to check whether our CDs properly scale, i.e., whether the time required by CDs to execute the coordination logic is affected or not when the number of clients inside the shop increases.

8 | EXPERIMENT RESULTS

This section reports on the experiment results that we obtained by repeating the experiment described in previous section 7 times. This permitted us to emulate every day of the week with different arrival rate of clients entering the shop. The average of the results confirm the method viability and solidity by showing that

i) the overhead due to CDs is negligible;

ii) the approach is scalable;

iii) undesired interactions are effectively prevented; and

iv) the network delay for exchanging coordination messages can be reduced.

TABLE 1 Results related to the presence in the shop of a varying number of clients over the course of 24 hours (scaled to minutes)

Time Slot (min)	Number of Choreography Instances (#instances)	Highest Parallelism Degree (#instances)	Coordination Logic Overhead (ms)	Coordination Messages Network Delay (ms)
00 – 08	100	13	0,41	24,68
08 – 12	500	44	0,41	35,65
12 – 14	300	28	0,33	30,42
14 – 16	400	32	0,35	32,03
16 – 18	800	112	0,29	33,00
18 – 20	1100	257	0,34	65,24
20 – 22	600	84	0,44	55,09
22 – 24	200	22	0,37	34,90

Table 1 distinguishes eight time slots, from 0 to 24 minutes. That is, by linearly scaling time (1 hour to 1 minute), in our experiment a day is emulated with a course of 24 minutes. Without loss of generality, this permitted us to considerably stress the system by concentrating in each time slot a large number of short-running, yet "highly-intensive", choreography instances running in parallel.

In the table, for each time slot, the `Number of Choreography Instances` column reports the total number of choreography instances executed during the time slot, hence the total number of clients that entered the shop during the time slot.

For each time slot, the `Highest Parallelism Degree` column reports the largest number of instances executing in parallel reached during the time slot, hence the largest number of clients simultaneously operating inside the shop in the considered timeframe. The highest parallelism degree reached during rush hours is 257.

Coordinating a specific interaction between two participants involves the time for exchanging the coordination messages across the network (i.e., network delay) plus the time for executing the coordination logic. Thus, for a participant P_x sending a message `m` to P_y, the coordination logic execution time is the time spent by $CD_{x.y}$ for executing the coordination algorithm upon receiving `m`. Consequently, the network delay is the time elapsed for exchanging the needed coordination messages across the network between $CD_{x.y}$ and all the other CDs it needs to synchronize with.

The `Coordination Logic Overhead` in Table 1 is then the average coordination time that is obtained through the arithmetic mean of the internal time spent by all CDs involved in the choreography for executing their coordination logic. The `Coordination Messages Network Delay` is the average network delay that is obtained through the arithmetic mean of the time required to exchange all the coordination messages for all CDs.

The trend of the `Coordination Logic Overhead` fluctuating in the narrow interval [0,29 ms – 0,44 ms], around the average time 0,37 ms (see Table 1). This is a very good result that can be summarized as follows:

Overhead *– the overhead due to CDs is negligible, meaning that the execution of the distributed coordination logic does not affect the choreography "performance" significantly.*

Scalability *– the approach is scalable, meaning that CDs suitably support a growing amount of coordination work with no degradation as the number of choreography instances running in parallel grows.*

Concerning scalability, it is important to clarify that we are not referring to (and it was not in the focus of our experiment) neither vertical scaling (increasing/reducing the computational resources, e.g., CPU and RAM available for the CD and service instances), nor horizontal scaling (augmenting/diminishing the number of CD and service replicas)[29]. In fact, as already said in Section 7.3, in our experiment setting, the same CD instances were shared by all choreography activations and their number was never scaled (and so it was for the number of service instances). Indeed, our goal was to evaluate the scalability of the distributed coordination logic performed by CDs (see Section 4) while managing an increasing number of different concurrent sessions – since different choreography instances can be in different states depending on the simultaneous presence in the

shop of increasing number of clients. It is more than clear that increasing the number of choreography instances, at a certain point, the available computational resources might be exhausted, and this is where vertical and horizontal scaling might come into play. This is however a different problem that concerns and can be solved with load balancing [29].

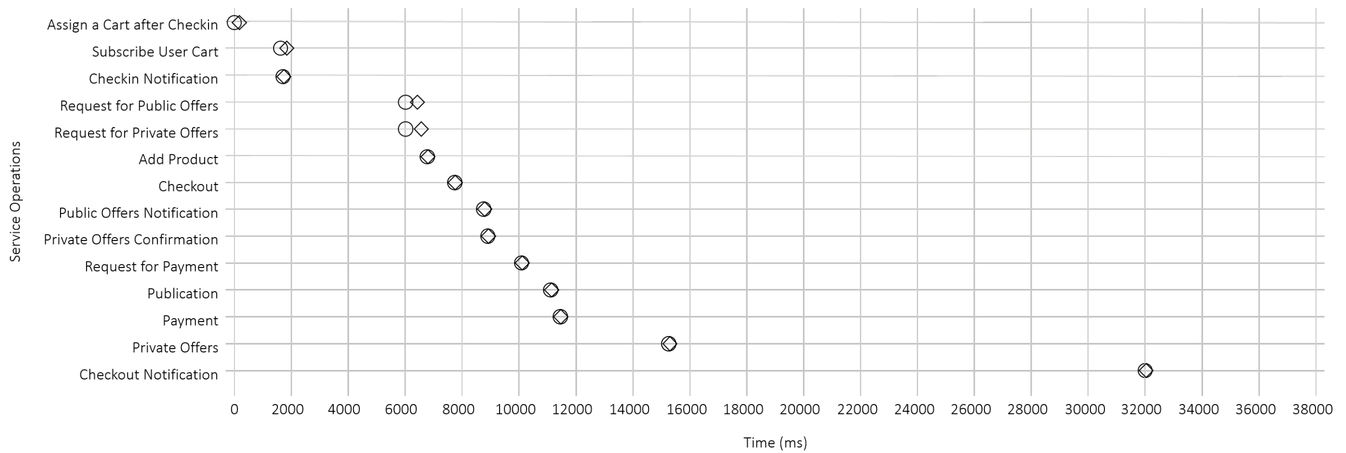

FIGURE 11 Average time for delegating service operations

One thing that leaps to the eye when looking at Table 1 is that the network delay has a considerable impact on the coordination time. Therefore, it stands to reason that an accurate deployment of CDs (that accounts for the deployment of the supervised services) can reduce the network delay if network characteristics and geographic distribution are suitably taken care of.

In the following, we provide a detailed analysis of the results summarized above. Figure 11 shows the overall delegation time for each operation. Specifically, each timeline highlights the time at which an initiating service sends a message (see the symbol ○ corresponding to the timestamp 1 in Figure 10) and the time at which the supervising CD actually forward the message to the receiving service (see the symbol ◇ corresponding to the timestamp 3 in Figure 10). Thus, within each timeline, the time distance between the symbols ○ and ◇ comprises not only the `Coordination Messages Network Delay` and the `Coordination Logic Overhead` but also the `Operation Waiting Time`, i.e., the time the CD waits for before being granted the right to proceed by the other involved CDs. It is worthwhile noticing that, for the majority of the operations, the messages are forwarded in a very short time. Rhombuses are in fact very close to the respective dots, except for the operations "Request for Public Offers" and "Request for Private Offers". In particular, with reference to the In-store Marketing and Sales choreography in Figure 2, the operation "Request for Public Offers" (resp., "Request for Private Offers") is invoked by the Marketing Application service which sends the message "public offers request" (resp., "private offers request") to the Marketing Manager service. As detailed in what follows, both operations are therefore coordinated by `CD-marketingapplication-marketingmanager`, which delays the forwarding of the two messages.

The histogram in Figure 12 shows more details about the time spent by `CD-marketingapplication-marketingmanager` to manage the "Request for Public Offers" and "Request for Private Offers" operations. The CD waits 39,31 ms before scheduling the "Request for Public Offers" operation and 52,63 ms before scheduling the "Request for Private Offers" operation (see `Operation Waiting Time` in the figure). This is due to the fact that the Marketing Application tried to invoke the Marketing Manager when the choreography was not in a correct execution state with respect to the forwarding of these messages. In fact, referring to the choreography specification in Figure 2, the tasks "Request for Public Offers" and "Request for Private Offers" have to be performed only after the tasks "Checkin Notification" and "Subscribe User Cart" are both accomplished, and the related CDs synchronize on the subsequent join point (i.e., `CD-shopentrance-marketingapplication` and `CD-client-smartcart` synchronize on the converging parallel gateway placed immediately after the parallel tasks "Checkin Notification" and "Subscribe User Cart").

In Figure 13, we can observe that the "Subscribe User Cart" and "Checkin Notification" tasks are executed in parallel. Since the "Subscribe User Cart" task is accomplished before, `CD-client-smartcart` needs to wait for joining with `CD-shopentrance-marketingapplication` (see the `CD Joining Time` in Figure 14).

Thus, as soon as the Marketing Application accomplishes the execution of "Checkin Notification", `CD-marketingapplication-marketingmanager` is ready to forward the "Request for Public Offers" and "Request for Private Offers" operations. However, before forwarding the two requests to the target service Marketing Manager, `CD-marketingapplication-marketingmanager` has to wait for receiving the coordination message sent by `CD-shopentrance-marketingapplication`, which will be sent to all CDs as soon as the "Checkin Notification" response is received. Waiting for coordination messages causes this negligible delay.

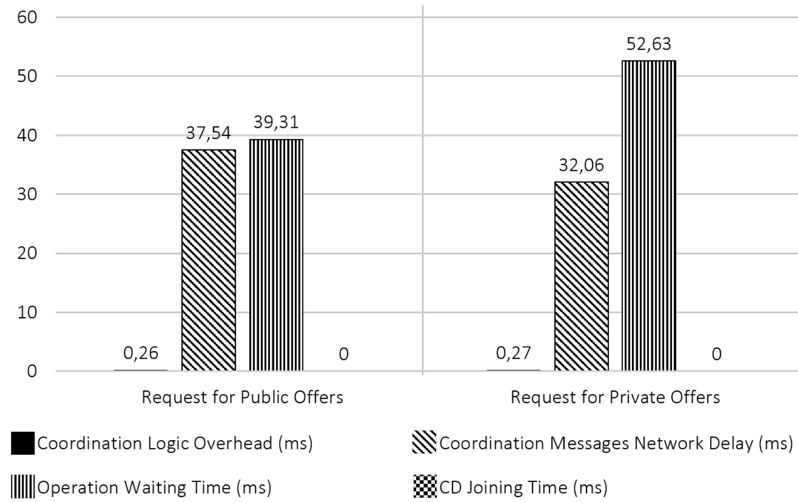

FIGURE 12 Average delegation time of `CD-marketingapplication- marketingmanager` when forwarding the operations "Request for Public Offers" and "Request for Private Offers"

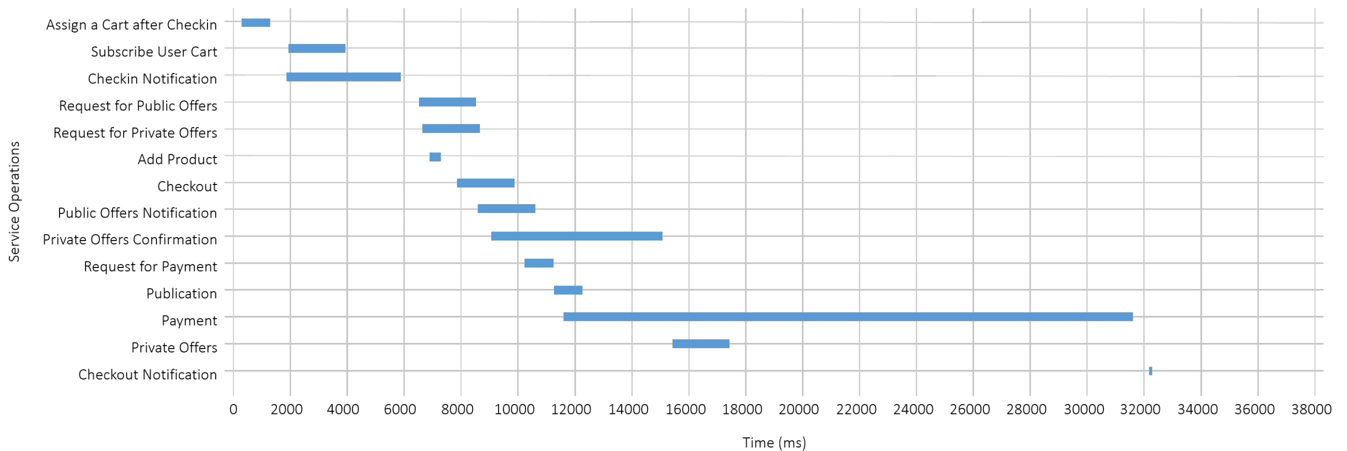

FIGURE 13 Average operations execution time

Still referring to Figure 14, note that the `CD Joining Time` spent by `CD-client-smartcart` is naturally due to the execution time of task "Checkin Notification". Moreover, the "Add Product" operation is not delayed, because Client is the initiating service of both "Subscribe User Cart" and "Add Product". Thus, it invokes the latter operation only after it has received the response to the former one (i.e., after `CD-client-smartcart` receives the coordination message from `CD-shopentrance-marketingapplication`).

For what concerns undesired interactions, we recall that the exchange of coordination messages also serve for preventing those service interactions that are not permitted by the choreography specification. With reference to the scenario described in Section 3, we recall that many undesired interactions can happen and all them were successfully prevented by our CDs. For instance, Figure 15 is dedicated to the undesired interaction with a malicious connotation only, and it concerns all the 400 malicious instances by reporting on their average delegation and discarding time. Still referring to Section 3, we recall that one such undesired interaction happens when the Client attempts to add further products into the cart (by performing the task "Add Product") after the payment of the previously added products (i.e., after the task "Payment" has been accomplished), and prior to exiting the shop (i.e., before performing "Checkout Notification"), hence avoiding to pay for them. In our experiment, we artificially forced the Client to have the just described "malicious" behavior in 400 choreography instances over 4000. Going on the merit, the task "Add Product" requires two message exchanges between the Client (which is the initiating participant) and the Smart Cart (which is the receiving participant). Therefore, `CD-client-smartcart` has to establish whether forwarding the request message to Smart Cart, or not. Figure 15 shows the timelines of one of the 400 malicious instances. After 32000 ms, the Client tries to invoke Smart Cart in order to add a further product (see the "Add Product

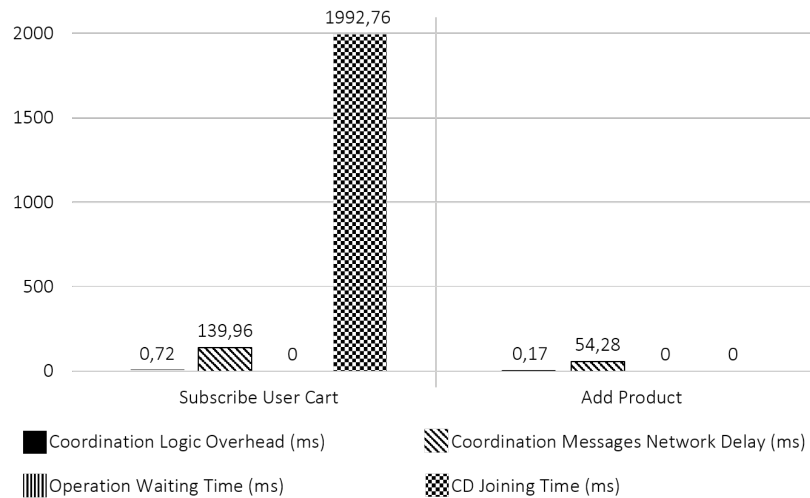

FIGURE 14 Average delegation time of `CD-client-smartcart` when forwarding the operations "Subscribe User Cart" and "Add Product"

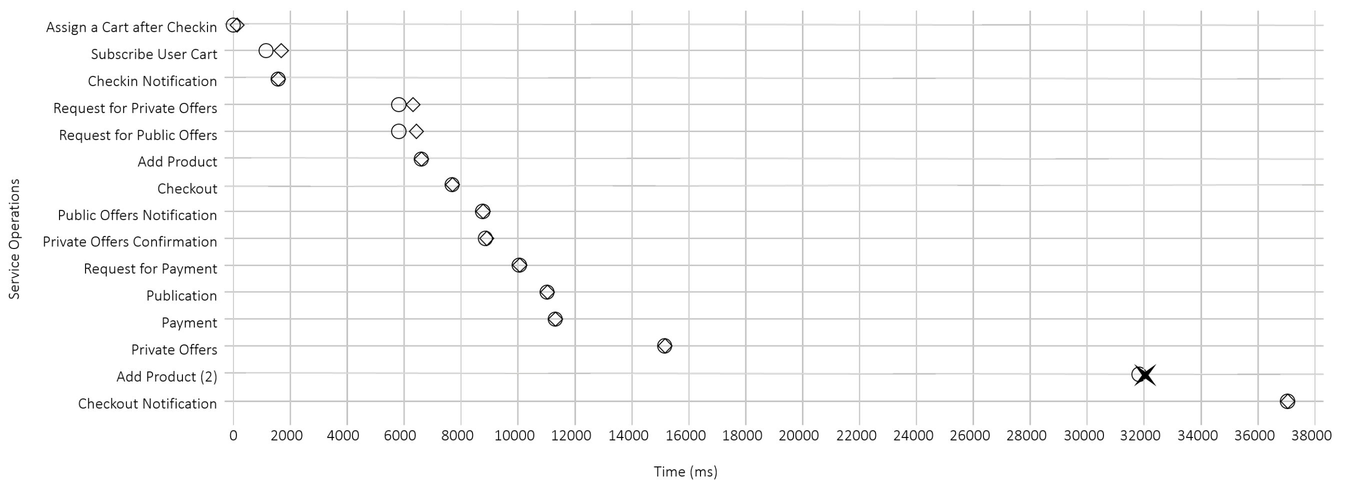

FIGURE 15 Average time for delegating service operations and discarding undesired interactions

(2)" operation). `CD-client-smartcart` recognizes that the choreography is in an execution state from where the forwarding of the message is not permitted and hence discards it (see the cross). It is worth noticing that, in order to fulfill the global collaboration prescribed by the choreography specification, in situations where a message cannot be immediately forwarded to the receiver, the CDs delays the execution of a given task by either putting the message in a wait-queue or discarding it if the message belongs to an undesired interaction.

Overall, the good result is that, thanks to the CDs, the choreographed system was able to prevent all sort of undesired interactions for the totality of the 4000 instances, beyond the (artificially caused) malicious ones for the 400 instances.

***Undesired interactions prevention** – all the undesired interactions are effectively prevented, meaning that CDs disallow those interactions that do not belong to the set of interactions allowed by the choreography, and can happen when the services collaborate in an uncontrolled way.*

Referring to Table 2, the *Coordination Logic Overhead* is 0,89 ms for `CD-client-smartcart` and 0,45 ms for `CD-shopentrance-marketingapplication`. This result shows that the portion of network delay due to the transmission of coordination messages (see the column "Coordination Messages Network Delay") has a relatively significant impact on the overall delegation time for both CDs. In fact, `CD-client-smartcart` and `CD-shopentrance-marketingapplication` spent 194,24 ms and 148,10 ms, respectively. Note also that these CDs exchange more coordination messages when compared to the other CDs. That is, beyond the messages exchanged to update the choreography state, these CDs need to also exchange coordination messages so as to join on the converging parallel gateway placed immediately after the tasks "Checkin Notification" and "Subscribe User Cart".

TABLE 2 Overall picture of the results for every CD

Coordination Delegate	Coordinated Service Operations (#operations)	Exchanged Coordination Messages (#messages)	Coordination Logic Overhead (ms)	Coordination Messages Network Delay (ms)	Operation Waiting Time (ms)	CDs Joining Time (ms)
CD-client-selfcheckoutmachine	1	1	0,14	33,58	0	0
CD-client-shopentrance	1	2	0,22	60,22	0	0
CD-client-smartcart	3	7	0,89	194,24	0	1992,76
CD-marketingapplication-instoretotem	1	1	0,25	30,82	0,77	0
CD-marketingapplication-marketingmanager	2	2	0,53	37,54	52,63	0
CD-marketingapplication-shoppingassistants	1	1	0,21	29,20	0,42	0
CD-marketingmanager-marketingapplication	2	2	0,36	51,94	0,89	0
CD-selfcheckoutmachine-marketingapplication	1	0	0,21	0	0,08	0
CD-selfcheckoutmachine-smartcart	1	1	0,23	34,66	0,97	0
CD-shopentrance-marketingapplication	1	5	0,45	148,10	0,86	0
CD-smartcart-selfcheckoutmachine	1	1	0,21	32,06	1,28	0

The network delay concerning the exchange of coordination messages network can be reduced by adopting different node allocation policies. In particular, the EE might be instructed for deploying these CDs in virtual machines that are geographically close, or in the same network, or in the same virtual machine in the best case. In our experiment, we first instructed the EE for deploying all the CDs by adopting a round-robin node allocation policy (see Section 7) without considering any deployment optimization aspect that, instead, could have been derived a priori from the choreography specification (the results reported so far refer to this policy). Then, attempting at reducing the network delay, we instructed the EE for deploying the `CD-shopentrance-marketingapplication` in the virtual machine `VM3`, which is the same used by `CD-client-smartcart` (Figure 8). This led to an 80% reduction of the network delay for the transmission of coordination messages between `CD-client-smartcart` and `CD-shopentrance-marketingapplication`, which spent 38,84 ms and 29,62 ms, instead of 194,24 ms and 148,10 ms, respectively. This is a good result that could be generalized to all cases as discussed in next section.

Reducing network delay – *the node allocation policies used by the enactment engine for deployment purposes can be acted upon to reduce the network delay for exchanging coordination messages.*

9 | PECULIARITIES AND LIMITATIONS

This section highlights some peculiarities of our approach and discusses limitations.

Peculiarities – As already introduced in Section 7.1, for each pair of participants involved in the same task, a CD is synthesized. It is worth to mention that the number of synthesized CDs is not dependent on the specified notation, i.e., BPMN2 choreography diagrams, nor on its completeness; rather, it is a realization/implementation choice. From a logical point of view, CDs can be seen as enhanced connectors that beyond communication allow for correct coordination. Moreover, maximum flexibility and modularity of the integration/coordination code implemented by CDs were our main requirements towards easy maintenance and evolvability of the system. In fact, as a coordination connector between participants P_x and P_y, $CD_{x,y}$ correctly coordinates the message exchanges from P_x to P_y. This information is inferred by the specified BPMN2 choreography diagram by looking at all the tasks with P_x as initiating participant. Note that, at the implementation level, we could have straightforwardly realized one CD_i per participant P_i by assembling the whole set of $CD_{i,1}, \ldots, CD_{i,n}$ ($n > 0$ and $n \neq i$) into a single CD. It supervises P_i by proxifies $P_1, \ldots, P_n$ in order to correctly coordinate the message exchanges from P_i to $P_1, \ldots, P_n$. Note also that, if we would have opted for realizing the CDs in this way, beyond maintenance and evolvability, we would have reduced not only the level of geographic distribution and, hence, real parallelism in terms of computation, but also the level of governance distribution in terms of responsability. For what concerns responsability, it must be noted that choreographies are particularly useful in those situations where multiple processes of different organizations need to collaborate but none of them want to, or cannot, takes full responsibility for performing the coordination of all (or part of the) interacting participant pairs. For example, this usually happens in Business-to-Business applications, which are cross-enterprise by definition and it is often the case that the involved parties have comparable negotiating or decisional power. Last but not least, the adopted solution enables a fully-distributed monitoring activity and eases fine-grained events logging (see Figure 10), as we do in our experimentation.

Limitations – Although the approach was conceived to be highly modular and fully distributed, the reference scenario considered in this paper suffers over-modularity due to the mono-directional nature of our CDs. In fact, by referring to the architecture description in Figure 7, in order to coordinate the interactions between `Smart Cart` and `Self Check-out Machine` and vice versa (resp., between `Marketing Application` and `Marketing Manager` and vice versa), the synthesis processor generates two different CDs: $CD_{4.3}$ and $CD_{3.4}$ (resp., $CD_{5.7}$ and $CD_{7.5}$), rather than a single one. When possible, synthesizing a single CD would reduce the overhead due to the exchanging of coordination messages and the network delay in situations where two choreography tasks that involve the same participants are performed sequentially. For example, referring to the choreography diagram in Figure 2, when the task "Request for Payment" is accomplished, `CD-selfcheckoutmachine-smartcart` notifies `CD-smartcart-selfcheckoutmachine` about the change of the choreography execution state so that `CD-smartcart-selfcheckoutmachine` allows the execution of the task "Payment". In this situation, a single CD coordinating these tasks would not send any notification, hence saving time. The same holds when these CDs need to synchronize on joining points that usually involve a higher number of coordination messages.

The enactment engine deploys CDs as prescribed by the produced `ChorSpec`, without considering any deployment optimization aspect. As discussed in Section 8, the enactment engine node allocation policies could be improved by deriving a set of deployment constraints out of the choreography specification. In this regard, a notion of correlation among CDs is of help. This would enable the introduction optimization mechanisms to reduce the latency of a request made by a client service toward another service. Specifically, the synthesis processor could identify those CDs that exchange a higher number of coordination messages and store this information in the architecture description. This information could then be used to derive a deployment strategy based on the notion CDs vicinity, which would reduce the latency for the transmission of coordination messages. For instance, by referring to our scenario, the synthesis processor could correlate `CD-client-smartcart` and `CD-shopentrance-marketingapplication` so to allow the enactment engine for considering this information at deployment time. Moreover, in addition to the correlation information, the EE could also consider the geographic distribution of the services and their interactions prescribed by the choreography specification. Inferring a suitable CDs allocation policy a priori, out of the choreography specification, is left as future work.

10 | RELATED WORK

This section organizes related work by distinguishing (i) papers using or proposing different specification methods, (ii) papers on concurrency and model checking, and (iii) papers proposing approaches to automated choreography realization.

Specification methods – There are many approaches aiming at specifying the services interactions in distributed systems [30,31,32,33,34,35]. In Particular, collaboration diagrams are a useful visual formalism for the specification of allowable conversations among the peers participating in a composite web service [30]. Message Sequence Charts (MSCs) provide another visual model for the specification of interactions in distributed systemss [36]. As opposed to the collaboration diagrams which only specify the ordering of send events, in the MSC model ordering of both send and receive events are captured. Montali *et al.* propose DecSerFlow, a declarative language, which uses Linear Temporal Logic and a SCIFF (a framework based on abductive logic programming), a formal specification of service interactions [31]. Other specification methods include OMG standard specification like SBVR (Semantic of Business Vocabulary and Rules) [32,33,34,35] and Choreography Diagrams, which are used by our approach.

Concurrency and model checking – The work in [?] proposes a choreography language equipped with a formal semantics that aims at supplying a formal framework for designing, analyzing and developing service-oriented computing systems. Specifically, the proposed choreography language permits to model the system as a set of conversations. The work in [37] introduces a static and dynamic semantics of two process calculi to capture the notion of compatibility in collaborations and choreographies. Global and end-point calculus are presented in order to describe the global and the local behavior of each choreography participant, respectively. This work leverage the notion of endpoint projection in order to check the conformance among endpoint processes. In our work, the choreography projection generates an expected behavioral model for each participant [5]. The model describes the participant expected behavior according to the flows of message exchanges specified by the choreography, in order to select a concrete service that plays the role of the (abstract) participant in the choreographed system. In [38], the authors explore in detail concurrency issues that can arise in choreography systems and propose a true-concurrency semantics for modeling the behavior of a long-running transaction that involves the interaction of multiple loosely-coupled online services. Moreover, the work in [38] points out that non-interleaving models have certain advantages over interleaving models (due to trace equivalence relations), and it goes into detail on how the purely distributed nature of service choreography has a bearing on the type of concurrency model considered. Bowles [36] presents a true-concurrency semantics based on *labelled prime event structures* to model multi-party conversations specified in message sequence charts (MSCs), by describing the sending and the receiving of messages as occurrences of events together with relations for expressing causal dependency and nondeterminism. In [39], *labelled prime event structures* are combined with vector languages to describe the behavior of components as understood in component-based software development. The notion of concurrency in [39] takes up on ideas from Mazurkiewicz trace languages [40] and it is based on an independence relation, that is events occurring on distinct component ports are independent and can happen concurrently, and events associated with the same component port are causally dependent and must be ordered in time. In [41] long-running transactions are captured with recovery and compensation

mechanisms for the underlying business services in order to ensure that a transaction either commits or is successfully compensated for. The work in[42] proposes a true-concurrent semantics for composing multiple sequence diagrams. A model-driven transformation is also proposed in order to transform sequence diagrams into Alloy models (a declarative textual modeling language based on first-order relational logic). Then, Alloy models are composed together in a single model with Alloy solver to represent all the possible behaviors derived by the composition, and for checking whether system properties are preserved in the composition. In[35], a compilation tool SBVR2Alloy is used to automatically validate and generate service choreographies specified in structured natural English language. Z3-SMT solver is used in[43]. Specifically, Z3 code is automatically generated from aspect weaving of scenario-based models, where aspects are given a true-concurrent semantics based on labelled event structures. In[5], we fully formalize the concurrency/coordination issues we are able to solve. Specifically, the work leverages the notion of "frontier", i.e., is able to automatically produce an intermediate representation of the choreography that, by transforming all the paired fork and join states, makes explicit all the possible interactions that can be performed through real parallelism. Summing up, the notion of frontier allows our approach to deal with a non-interleaving model. Furthermore, shifting from verification to synthesis, our approach goes beyond conformance and realizability checking by performing choreography realizability enforcement.

Automated choreography realization – Chakraborty and Mukund propose an approach to enforce synchronizability and realizability of a choreography[44]. Their method is able to automatically generate monitors, which behave as local controllers interacting with their peers and the rest of the system to guarantee that the peers will comply with the choreography specification. Our conceptualization of CD is "similar" to the concept of monitor used by these authors[44]. However, the two synthesis methods are different. Chakraborty and Mukund use an iterative process to generate the monitors[44], automatically refining their behavior. Initially, they are obtained by generating the set of peers via choreography projection. Then, equivalence checking is employed for automatically checking the system synchronizability and realizability. In case of synchronizability or realizability violation, the generated counterexample acts as an input for the process of improving the monitors with a new synchronization message. Our work is different since our approach is reuse-based. We synthesize CDs that enable the choreography realization from concrete services that already exist, allowing development teams to develop business services without worrying with the CDs existence. As detailed in our previous work[5], our service selection process is not based on the exact matching of behavior, allowing for selected services to present a refinement of the behavior of corresponding abstract participants. Furthermore, Chakraborty and Mukund report on few experiments[44] – carried on artificial examples – that aim to show the impact of increasing the number of choreography peers in terms of the space complexity of the generated coordination logic and its overall execution time, only. Farah *et al.* address realizability based on a priori verification techniques, employing refinement and proof-based formal methods[45]. This approach is different from ours since it relies on peer projection that is correct by construction, which proscribe the use of already existing services. On the other hand, our approach realizes choreograph specifications by reusing external peers (possibly black-box), rather than generating them. Dealing with black-box services is a requirement for us, considering that most of developers will not change services that are already working or change the service technology to comply with the choreography tooling.

The ASTRO toolset supports automated composition of Web services and the monitoring of their execution[46]. It aims to compose a service starting from a business requirement and the description of the protocols defining available external services. Unlike our approach, ASTRO deals with centralized orchestration-based business processes rather than fully decentralized choreography-based ones.

Carbone and Montesi present a unified framework for choreography development based on correct construction by preventing deadlock and ensuring communication safety[12]. Lanese *et al.* discuss the AIOCJ choreography language, which allows for developing adaptable choreographies[13]. Differently from our proposal, these related works[12,13] focus more on developing choreographies from scratch rather than realizing them through the reuse of existing participants. Nonetheless, they address adaptability and flexibility[47], spiking re-synthesis when a change occurs. The scope of this paper does not include adaptability and flexibility.

Montesi *et al.* exploit multiparty session types to automatically translate choreographies into local specifications of the communications that each endpoint should implement[48,49]. Choreographic programming is a different approach prototyped by Montesi in 2013[50] and further developed in the recent years by his research group[51,52,53,54,55]. The code for all endpoints is written in a single choreography, which instructs both communications and how data should be computed. Then, the choreography is automatically translated into an executable program for each endpoint. Differently from the works on multiparty session types discussed in[48,49], choreographic programming does not require a code checker for mainstream programs, because correctness is ensured by a single theorem that establishes a semantic correspondence between the choreography and the programs generated from it (correctness by construction). However, the disadvantage of choreographic programming is that the implementation details of all endpoints must be shared in a single choreography, which breaks modular software development. The benefits of choreographic programming are lost when dealing with multiple choreographies.

11 | CONCLUDING DISCUSSION

This paper presented an integrated development and run-time platform for the realization of choreography-based systems, which supports all the development activities, including specification, code synthesis, automatic deployment, enactment, and monitoring on the Cloud.

We reported on the results of the experiment we conducted with a use case in the in-store marketing and sales domain, and discuss the lesson learned and limitations of the approach. Strictly concerning the purpose of the experiment, we focused on the the two main components of the platform, namely, the *synthesis processor* and the *enactment engine*, which offer automatic choreography code generation facilities, and deployment & enactment facilities, respectively. The experiment demonstrated that the approach is viable and the platform can be effectively exploited in practical contexts by showing how the interplay of synthesis processor and the enactment engine enables the systematic transition from service choreography design to execution.

In the following, we articulate a discussion on three main aspects that, although not treated in this paper, concern our ongoing research work and related achievements, namely, *heterogeneous interaction protocols*, *security*, *microservices*, *market take-up and implementation enhancement*, and *platform evaluation*.

Heterogeneous interaction protocols – Beyond pure coordination, ongoing work concerns dealing with heterogeneous interaction protocols and, hence, with protocol adaptation that aims to solve mismatches at the level of service operations and related I/O parameters. This calls for supporting data-based coordination through the elicitation and application of complex data mappings to match the data types of messages sent or received by mismatching participant services. This means effectively coping with heterogeneous service interfaces and dealing with as much Enterprise Integration Patterns [56] and Protocol Mediation Patterns [57] as possible, in a fully automatic way. In this direction, we already achieved promising results in previous work [58,59,60,61].

Security – In the practice of choreography development, the participant services can belong to different security domains governed by different authorities and use different identity attributes that are utilized in their access control policies. Main concerns to be addressed include: (i) users and services belonging to an organization A willing to access a service in an organization B should be accepted by B with their identity in A; (ii) users or services having different attributes in A and willing to access a service in B should be able to use their attributes in A to check their rights in B; (iii) service-middleware interaction must be secured to avoid attacks (e.g., injecting malicious services as dependencies within the choreography). Our approach is being enhanced to provide fine-grained per-service identity management methods for cross-federation security enforceability at the service level. These methods, together with related deployment and enactment mechanisms, serve to filter the service behavior according to an associated identity profile, up to the granularity level of a single service operation. Thus, in addition to the functional roles specified by the choreography model, security-aware roles should be specified so to filter the functional roles on security requirements.

Microservices – In the current trend towards microservice-based architectures, the choreography principles have been adopted. Lewis and Fowler describe microservices as an architectural style [62] in which a single application is composed of many small units that are built around business capabilities, are independently deployable and interact through the network [63]. Such network interaction takes place in a choreographed manner, since each participant knows precisely with whom and when to interact. Moreover, enforcing a specified global coordination logic across hundreds of in-house microservices built by several teams is a non-trivial endeavor since there is no current de facto standard for this. Imposing a specific technology for generating code that complies with the choreography might not be well accepted by developers since an advantage for microservices acknowledged by practitioners is precisely the freedom in exploring new technologies [64]. The correctness of the coordination logic is, thus, weakly guaranteed by every engineer crafting each microservice. Current microservices systems are composed of mostly in-house services, while the few external services currently do not present a significant impact on coordination logic. This happens because in-house services, usually, interact with third-party services in a simple request-response pattern, i.e., external services usually do not actively invoke in-house services. Moreover, whether the third-party service needs to invoke other services behind the scenes, it is not a concern for the microservices system owner. However, today's Internet provides billions of ready-to-use services and promotes the possibility of more complex patterns of interaction among in-house and third-party services becoming commonplace. In-house services can, for example, register handler endpoints for external services send notifications about events of the business flow (e.g., the Post Office service notifies a letter is delivered). In this context, enforcing the agreed cross-organizational interaction patterns – the choreography – is a more complicated issue. As future work, we plan to extend the proposed approach to also consider microservices as choreography participants.

Market take-up and implementation enhancement – To enable the market take-up and further enhancement of the synthesis processor implementation and the enactment engine implementation by third-party developers, especially Small Medium Enterprises, including development of new applications to be commercialized, we released the code under the umbrella of the OW2 Consortium[9] and the Future Internet Software and Services initiative (FISSi[10]). These initiatives promote OW2 Future Internet software towards members and non-members alike, open-source vendors

[9] https://projects.ow2.org/view/chorevolution/
[10] www.ow2.org/view/Future_Internet

and proprietary vendors alike, with a market-oriented approach. A desirable achievement for the near future would be to establish a community of developers and third-party market stakeholders (e.g., users, application vendors, policy makers, etc.) around our approach.

Platform evaluation – The results of the conducted experiment confirm confidence in the approach, and show that the platform can be applied in practical contexts. Indeed, in [65], we also evaluated the proposed approach against two other industrial use cases to understand how effective and how efficient is the platform in helping the implementation of service choreographies. The evaluation involved two IT companies, one in Italy and the other one in Sweden. During the evaluation, the developers of both the companies experienced a significant time decrease with respect to their daily development approaches. The results of the experiments indicated that the platform has a great potential in developing choreography-based applications and the two use cases got a full benefit from it. As future work, more pilots and development cases will allow us to consolidate the technical maturity of the product and then pose the basis for a commercial validation.

ACKNOWLEDGEMENTS

The authors would like to thank the CHOReOS project (FP7 European program #FP7-ICT-2009-5) and FAPESP proc. 14/50937-1, and FAPESP proc. 15/24485-9 for their support.

References

1. European Commission . Next Generation Internet (NGI) - Re-think the Internet (Dec. 6, 2018 update). 2018.

2. Seok S, Muhammad A, Kim K, et al. multiFIA: Multi-dimensional Future Internet Architecture. In: *Proc. Ubiquitous and Future Networks*. ; 2017: 87-92

3. European Commission . Digital Agenda for Europe - Next Generation Internet initiative (Sept. 10, 2018 update). 2018.

4. Güdemann M, Poizat P, Salaün G, Ye L. VerChor: A Framework for the Design and Verification of Choreographies. *Trans. on Services Computing* 2016; 9(4): 647-660. doi: 10.1109/TSC.2015.2413401

5. Autili M, Inverardi P, Tivoli M. Choreography Realizability Enforcement through the Automatic Synthesis of Distributed Coordination Delegates. *Science of Computer Programming* 2018; 160: 3 - 29. doi: https://doi.org/10.1016/j.scico.2017.10.010

6. Chen L, Englund C. Choreographing Services for Smart Cities: Smart Traffic Demonstration. In: *85th IEEE Vehicular Technology Conference*. IEEE; 2017: 1–5

7. Lemos AL, Daniel F, Benatallah B. Web Service Composition: A Survey of Techniques and Tools. *ACM Computing Surveys* 2016; 48(3): 33:1–33:41.

8. Corradini F, Fornari F, Polini A, Re B, Tiezzi F. A formal approach to modeling and verification of business process collaborations. *Science of Computer Programming* 2018; 166: 35 - 70.

9. Poizat P, Salaün G. Checking the Realizability of BPMN 2.0 Choreographies. In: *Proc. 27th Symposium on Applied Computing*. ACM; 2012: 1927–1934

10. Basu S, Bultan T, Ouederni M. Deciding Choreography Realizability. In: *Proc. 39th Symposium on Principles of Programming Languages*. ACM; 2012: 191–202

11. Autili M, Ruscio DD, Salle AD, Inverardi P, Tivoli M. A Model-Based Synthesis Process for Choreography Realizability Enforcement. In: *16th Int. Conf. on Fundamental Approaches to Software Engineering*. Springer; 2013: 37–52

12. Carbone M, Montesi F. Deadlock-freedom-by-design: multiparty asynchronous global programming. In: *Proc. 40th Symposium on Principles of Programming Languages*. ACM; 2013: 263–274.

13. Lanese I, Montesi F, Zavattaro G. The Evolution of Jolie: From Orchestrations to Adaptable Choreographies. In: *Software, Services, and Systems*. Springer; 2015: 506–521.

14. Basu S, Bultan T. Choreography conformance via synchronizability. In: *Proc. Int. Conf. on World Wide Web*. ACM; 2011.

15. Basu S, Bultan T. Automated Choreography Repair. In: *Proc. 19th Int. Conf. on Fundamental Approaches to Software Engineering*. Springer; 2016: 13–30

16. Autili M, Salle AD, Perucci A, Tivoli M. On the Automated Synthesis of Enterprise Integration Patterns to Adapt Choreography-based Distributed Systems. In: *Int. Workshop on Foundations of Coordination Languages and Self-Adaptive Systems*. ; 2015

17. Foster H, Uchitel S, Magee J, Kramer J. Model-Based Analysis of Obligations in Web Service Choreography. In: *Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services (AICT-ICIW'06)*. ; 2006: 149-149.

18. Autili M, Inverardi P, Tivoli M. Automated Synthesis of Service Choreographies. *IEEE Software* 2015; 32(1): 50–57. doi: 10.1109/MS.2014.131

19. Leite L, Moreira CE, Cordeiro D, Gerosa MA, Kon F. Deploying large-scale service compositions on the cloud with the CHOReOS Enactment Engine. In: *Proc. 13th IEEE Int. Symposium on Network Computing and Applications*. IEEE Computer Society; 2014: 121–128.

20. Autili M, Tivoli M. Distributed Enforcement of Service Choreographies. In: *Proc. 13th Int. Workshop on Foundations of Coordination Languages and Self-Adaptive Systems*. ; 2015: 18–35

21. Lamport L. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 1978; 21(7): 558–565. doi: 10.1145/359545.359563

22. Allen R, Garlan D. A Formal Basis for Architectural Connection. *ACM Transaction on Software Engineering Methodologies* 1997; 6(3): 213âĂŞ249. doi: 10.1145/258077.258078

23. Hamilton J. On Designing and Deploying Internet-Scale Services. In: *Proc. 21st Large Installation System Administration Conference (LISA '07)*. USENIX; 2007.

24. Humble J, Molesky J. Why Enterprises Must Adopt Devops to Enable Continuous Delivery. *Cutter IR Journal, The Journal of Information Technology Management* 2011; 24(8): 6–12.

25. Dolstra E, Bravenboer M, Visser E. Service configuration management. In: *Proc. 12th Intl. Workshop on Software Configuration Management*. ACM; 2005.

26. Fowler M. Inversion of Control Containers and the Dependency Injection pattern. 2004. http://martinfowler.com/articles/injection.html.

27. Humble J, Farley D. *Continuous Delivery*. Addison-W. . 2011.

28. Tavis M, Fitzsimons P. Web Application Hosting in the AWS Cloud: Best Practices. tech. rep., Amazon; 2012.

29. Autili M, Perucci A, De Lauretis L. *A Hybrid Approach to Microservices Load Balancing*: 249–269; Cham: Springer International Publishing . 2020

30. Bultan T, Fu X. Specification of realizable service conversations using collaboration diagrams. *Service Oriented Computing and Applications* 2008; 2(1): 27–39. doi: 10.1007/s11761-008-0022-7

31. Montali M, Pesic M, Aalst v. dWMP, Chesani F, Mello P, Storari S. Declarative specification and verification of service choreographiess. *ACM Trans. Web* 2010; 4(1): 3:1–3:62. doi: 10.1145/1658373.1658376

32. Afreen H, Bajwa IS, Bordbar B. SBVR2UML: A Challenging Transformation. In: *2011 Frontiers of Information Technology, FIT 2011, Islamabad, Pakistan, December 19-21, 2011*. IEEE Computer Society; 2011: 33–38

33. Bajwa IS, Lee MG, Bordbar B. SBVR Business Rules Generation from Natural Language Specification. In: *AI for Business Agility, Papers from the 2011 AAAI Spring Symposium, Technical Report SS-11-03, Stanford, California, USA, March 21-23, 2011*. AAAI; 2011.

34. Manaf NA, Moschoyiannis S, Krause PJ. Service Choreography, SBVR, and Time. In: Proença J, Tivoli M., eds. *Proceedings 14th International Workshop on Foundations of Coordination Languages and Self-Adaptive Systems, FOCLASA 2015, Madrid, Spain, 5th September 2015*. . 201 of *EPTCS*. ; 2015: 63–77

35. Manaf NA, Antoniades A, Moschoyiannis S. SBVR2Alloy: An SBVR to Alloy Compiler. In: *10th IEEE Conference on Service-Oriented Computing and Applications, SOCA 2017, Kanazawa, Japan, November 22-25, 2017*. IEEE Computer Society; 2017: 73–80

36. Filipe JK. Modelling concurrent interactions. *Theor. Comput. Sci.* 2006; 351(2): 203–220. doi: 10.1016/j.tcs.2005.09.068

37. Carbone M, Honda K, Yoshida N, Milner R, Brown G, Ross-Talbot S. A theoretical basis of communication-centred concurrent programming. tech. rep., ; 2006.

38. Moschoyiannis S, Krause PJ. True Concurrency in Long-running Transactions for Digital Ecosystems. *Fundamenta Informaticae* 2015; 138(4): 483–514. doi: 10.3233/FI-2015-1222

39. Filipe JK, Moschoyiannis S. Concurrent Logic and Automata Combined: A Semantics for Components. *Electron. Notes Theor. Comput. Sci.* 2007; 175(2): 135–151. doi: 10.1016/j.entcs.2007.03.008

40. Mazurkiewicz AW. Basic notions of trace theory. In: Bakker dJW, Roever dWP, Rozenberg G., eds. *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop, Noordwijkerhout, The Netherlands, May 30 - June 3, 1988, Proceedings*. . 354 of *Lecture Notes in Computer Science*. Springer; 1988: 285–363

41. Bowles JKF, Moschoyiannis S. When Things Go Wrong: Interrupting Conversations. In: Fiadeiro JL, Inverardi P., eds. *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. . 4961 of *Lecture Notes in Computer Science*. Springer; 2008: 131–145

42. Alwanain M, Bordbar B, Bowles JKF. Automated Composition of Sequence Diagrams via Alloy. In: Pires LF, Hammoudi S, Filipe J, Neves dRC. , eds. *MODELSWARD 2014 - Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development, Lisbon, Portugal, 7 - 9 January, 2014*. SciTePress; 2014: 384–391

43. Bowles JKF, Bordbar B, Alwanain M. Weaving True-Concurrent Aspects Using Constraint Solvers. In: Desel J, Yakovlev A., eds. *16th International Conference on Application of Concurrency to System Design, ACSD 2016, Torun, Poland, June 19-24, 2016*. IEEE Computer Society; 2016: 35–44

44. Güdemann M, Salaün G, Ouederni M. Counterexample Guided Synthesis of Monitors for Realizability Enforcement. In: Chakraborty S, Mukund M., eds. *Automated Technology for Verification and Analysis*. LNCS. Springer. 2012 (pp. 238-253).

45. Farah Z, Ait-Ameur Y, Ouederni M, Tari K. A correct-by-construction model for asynchronously communicating systems. *Software Tools for Technology Transfer* 2016; 19(4): 465–485.

46. Trainotti M, Pistore M, Calabrese G, et al. ASTRO: Supporting Composition and Execution of Web Services. In: *ICSOC'05*. Springer. 2005 (pp. 495-501, LNCS 3826).

47. Reichert M, Weber B. *Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies* . 2012.

48. Carbone M, Lindley S, Montesi F, Schürmann C, Wadler P. Coherence Generalises Duality: a logical explanation of multiparty session types. In: *27 International Conference on Concurrency Theory (CONCUR'16)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik; 2016.

49. Carbone M, Montesi F, Schürmann C, Yoshida N. Multiparty session types as coherence proofs. *Acta Informatica* 2017; 54(3): 243–269.

50. Montesi F. Choreographic Programming. 2013. http://www.fabriziomontesi.com/files/choreographic_programming.pdf. Accessed 31 January 2020.

51. Carbone M, Cruz-Filipe L, Montesi F, Murawska A. Multiparty Classical Choreographies. In: *Logic-Based Program Synthesis and Transformation*. Springer International Publishing; 2019: 59–76.

52. Carbone M, Montesi F, Schürmann C. Choreographies, Logically. *Distributed Computing* 2018; 31(1): 51–67.

53. Cruz-Filipe L, Montesi F. A Core Model for Choreographic Programming. In: *Formal Aspects of Component Software*. Springer International Publishing; 2017: 17–35.

54. Cruz-Filipe L, Montesi F. Procedural Choreographic Programming. In: *Formal Techniques for Distributed Objects, Components, and Systems*. Springer International Publishing; 2017: 92–107.

55. Cruz-Filipe L, Montesi F, Peressotti M. Communications in choreographies, revisited. In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. Association for Computing Machinery; 2018: 1248–1255.

56. Hohpe G, Woolf B. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions - 15th printing 2011*. Addison-Wesley Longman Publishing Co., Inc. . 2004.

57. Li X, Fan Y, Wang J, Wang L, Jiang F. A Pattern-Based Approach to Development of Service Mediators for Protocol Mediation. In: *Proc. 7th Working IEEE/IFIP Conference on Software Architecture*. WICSA. IEEE Computer Society; 2008: 137-146

58. Inverardi P, Tivoli M. Automatic Synthesis of Modular Connectors via Composition of Protocol Mediation Patterns. In: *Proc. 35th Int. Conf. on Software Engineering*. ICSE. IEEE Press; 2013: 3–12

59. Autili M, Di Salle A, Gallo F, Pompilio C, Tivoli M. On the Model-driven Synthesis of Evolvable Service Choreographies. In: *Proc. 12th European Conf. on Software Architecture*. ACM; 2018: 1–6.

60. Autili M, Inverardi P, Spalazzese R, Tivoli M, Mignosi F. Automated synthesis of application-layer connectors from automata-based specifications. *Journal of Computer and System Sciences* 2019. doi: https://doi.org/10.1016/j.jcss.2019.03.001

61. Autili M, Di Salle A, Gallo F, Pompilio C, Tivoli M. Model-driven adaptation of service choreographies. In: *Proc. 33th Symposium on Applied Computing*. ACM; 2018: 1441–1450.

62. Fielding RT. *Architectural styles and the design of network-based software architectures*. PhD thesis. University of California, 2000.

63. Lewis J, Fowler M. Microservices, a definition of this new architectural term. 2014. https://martinfowler.com/articles/microservices.html. Accessed 31 January 2020.

64. Soldania J, Tamburrib DA, Heuvel WJVD. The pains and gains of microservices: A Systematic grey literature review. *Journal of Systems and Software* 2018; 146: 215–232.

65. Autili M, Di Salle A, Gallo F, Pompilio C, Tivoli M. CHOReVOLUTION: Automating the Realization of Highly–Collaborative Distributed Applications. In: *Coordination Models and Languages*. Springer International Publishing; 2019: 92–108.

How to cite this article: M. Autili, A. Perucci, L. Leite, M. Tivoli, F. Kon, and A. Di Salle (2020), Highly-collaborative distributed systems: synthesis and enactment at work, *Concurrency and Computation: Practice and Experience*, e6039. https://doi.org/10.1002/cpe.6039