Computing Without Borders: The Way Towards Liquid Computing

Computing Without Borders: The Way Towards Liquid Computing

Marco Iorio, Fulvio Risso, Alex Palesandro, Leonardo Camiciotti, and Antonio Manzalini

***Abstract*—Despite the de-facto technological uniformity fostered by the cloud and edge computing paradigms, resource fragmentation across isolated clusters hinders the dynamism in application placement, leading to suboptimal performance and operational complexity. Building upon and extending these paradigms, we propose a novel approach envisioning a transparent continuum of resources and services on top of the underlying fragmented infrastructure, called *liquid computing*. Fully decentralized, multi-ownership-oriented and intent-driven, it enables an overarching abstraction for improved applications execution, while at the same time opening up for new scenarios, including resource sharing and brokering. Following the above vision, we present *liqo*, an open-source project that materializes this approach through the creation of dynamic and seamless Kubernetes multi-cluster topologies. Extensive experimental evaluations have shown its effectiveness in different contexts, both in terms of Kubernetes overhead and compared to other open-source alternatives.**

***Index Terms*—Computing Continuum, Cloud/Edge Computing, Task Offloading, Inter-Cluster Network Fabric, Liquid Computing**

I. Introduction

In the last years, containerization has increasingly gained popularity as a lightweight solution to package applications in an interoperable format [1], independently of the target infrastructure. This uniform substratum paved the way for the cloud native revolution, with novel applications shifting their focus from single servers to entire data centers, and where dedicated orchestrators manage the lifecycle of microservice applications. As of today, Kubernetes emerged as the de-facto open-source framework for container orchestration, bridging the semantic gaps across competing infrastructure providers [2]. With the rise of the edge and fog computing paradigms [3]–[5] as solutions accounting for geographical closeness, reduced latency and improved privacy, the same approaches are being progressively extended towards smaller data centers at the network border, benefiting from uniform primitives to foster service agility.

Despite the emergence of common interfaces for applications orchestration being key towards a real *edge to cloud continuum* [6], [7], industry-standard approaches handle each infrastructure as a multitude of (connected) isolated silos instead of a unique virtual space. This leads to a sub-optimal fragmented view of the overall available resources, preventing the seamless deployment of fully distributed applications. Indeed, edge data centers cannot depend on a single centralized control plane, for resiliency (i.e., preventing failure propagation in case of network partitioning) and performance reasons, as orchestration platforms typically suffer if nodes are geographically spread over high-latency WANs [8]–[10]. Besides the edge landscape, resource fragmentation affects also larger data centers, with many companies increasingly witnessing the cluster sprawl phenomenon [11], [12]. This trend finds its roots in scalability concerns, in the hybrid-cloud (i.e., the combination of on-premise and public cloud) and multi-cloud approaches [13], which aim for high availability, geographical distribution and cost-effectiveness, while granting access to the breadth of capabilities offered by competing cloud providers. Additionally, non-technical requirements such as law regulations, mergers and acquisitions, physical isolation policies and separation of concerns contribute to the proliferation of clusters.

Fragmentation also hinders the potential dynamism in the workload placement [14]–[16], forcing each application to be assigned upfront to a specific infrastructure. No resource compensation is ever possible, hence preventing jobs from transparently moving from an overloaded cluster, e.g., due to unexpected spikes of requests, to another one, underused and potentially offering better performance. At the same time, the deployment of complex applications composed of multiple microservices, each one with specific requirements (e.g., low latency, high computational power, access to specialized hardware, ...), as well as the enforcement of proper geographical distribution and high-availability policies, requires the interaction with different infrastructures. However, this prevents to rely on the single point of control abstraction, which would allow to coordinate the deployment of arbitrarily complex applications across the entire resource continuum, no matter how many nodes and clusters it is composed of.

Accounting for these demands, in this paper we advocate the opportunity for a novel architectural paradigm: *liquid computing*[1], which builds upon and extends the well-established cloud and edge computing approaches towards an endless *computing continuum*. Then, we present a first real implementation of a software framework enabling a continuum of computational resources and ready-to-consume services

M. Iorio and F. Risso are with Politecnico di Torino, Torino, Italy. E-mail: {name}.{surname}@polito.it. A. Palesandro was with Politecnico di Torino at the time of this work. L. Camiciotti is with Consorzio TOP-IX, Torino, Italy. A. Manzalini is with TIM, Torino, Italy.

This work has been published in IEEE Transactions on Cloud Computing: https://doi.org/10.1109/TCC.2022.3229163.

[1]This term was first coined in 2014 by InfoWorld [17] as a synonym of pervasive computing, i.e., the capability of keep working on a given task across multiple devices such as PCs and tablets. This paper refers to a broader concept, which encompasses the creation of a resource continuum composed of cloud and edge infrastructures, on-premise clusters, as well as, in its widest form, single end-user and IoT devices.

spanning across multiple physical infrastructures. Overall, the resulting computing domain abstracts away the specificity of each cluster, presenting to the final users, either actively participating as actors or simply renting off-the-shelf services, a *unique* and *borderless* pool of available resources, the so-called *big cluster*. Thanks to this abstraction, applications are no longer constrained in a specific silo, but free to *fly* in the entire infrastructure, selecting the most appropriate location depending on its requirements (e.g., a user facing service may be replicated at the edge to account for low latency, while another might be constrained to European infrastructures to comply with GDPR), and the available resources, while retaining full compatibility (hence, models, tools, and commands) with vanilla Kubernetes.

The remainder of the paper is organized as follows. Section II discusses our liquid computing vision, with its key pillars detailed in Section III. Section IV describes the main characteristics of *liqo*[2], an open-source project which fosters this idea by enabling dynamic and seamless Kubernetes multi-cluster topologies, with the most relevant implementation aspects detailed in Section V. Section VI presents its experimental evaluation, while Section VII discusses related approaches. Finally, Section VIII draws the main conclusions.

II. The Liquid Computing Vision

We envision liquid computing as a *continuum of resources and services* allowing the seamless and efficient deployment of applications, independently of the underlying infrastructure. We present here the main characteristics of liquid computing, followed by the most significant deployment scenarios.

A. Main characteristics

We believe this paradigm shall be composed of the following four distinguishing characteristics.

1) Intent-driven: A consumer can assign to each workload the desired execution constraints through high-level policies, without knowing about the infrastructural details. Overall, liquid computing brings the *cattle service model* [18] to a greater scale. Similarly to servers in a data center, with no one caring about where each task is executed, as long as requirements are fulfilled, this paradigm blurs the cluster borders so that users are relieved from selecting a specific infrastructure for their applications. Yet, different clusters are definitely associated with different properties (e.g., in terms of geographical location and security characteristics) and, indeed, this is one of the main driving reasons behind cluster sprawling. Thus, it is of utmost importance the adoption of an *intent-driven* approach, allowing final users to enrich each workload with a set of high-level policies to express the associated constraints (e.g., geographical locality and spreading, costs, capabilities, ...); automated schedulers shall select the best execution place across the entire border-less infrastructure, depending on the available resources and enforcing in concert the user-specified policies. Yet, we deem at the same time the resource continuum abstraction to enable more contextualized scheduling decisions (given the knowledge about the entire infrastructure), allowing for further optimizations and better scalability compared to the siloed approach.

2) Decentralized architecture: The resource continuum stems from a peer-to-peer approach, with no central point of control and management entities, as well as no intrinsically privileged members. Following a decentralized and peer-based model like the Internet, the liquid computing approach fosters the coexistence of multiple actors, including larger cloud providers, smaller, territory-linked enterprises and possibly even small office/home owners. Indeed, each entity can autonomously and dynamically decide who to peer with (hence, share resources), similarly to the concept of *Autonomous Systems* in the Internet inter-domain routing. A dynamic *discovery and peering* protocol is in charge of the automatic identification of available peers and the negotiation of peering contracts based on the demands and offers of each actor, along with their specific constraints; optionally, the above operations could also be delegated to an intermediate dedicated entity such as a *broker*. No sensitive information disclosure is mandated (e.g., infrastructural setup), with the entire process possibly involving only the request for a certain amount of abstract resources (e.g., CPU, memory, ...) and the offer of available ones, together with the associated cost. Besides peering establishment, decentralization also concerns the preserved ability of each cluster to evolve independently, thanks to the local orchestration logic, and the support for the creation of arbitrary topologies, with different points of entry for the deployment of different workloads.

3) Multi-ownership: Each actor maintains the full control of his own infrastructure, while deciding at any time how many resources and services to share and with whom. Although single clusters are expected to be under the control of a single entity, the entire *resource ocean* would likely span across different administrative domains. Once a new peering is established, the control plane of the target infrastructure is in charge of configuring the appropriate isolation primitives (e.g., resource quota, network and security policies, ...), based on the underlying orchestration capabilities, to enforce the shared resource slice and prevent noisy neighbors phenomena. Specifically, we foresee a *shared security responsibility model*, with the provider responsible for the creation of well-defined sandboxes and the possible provisioning of additional security mechanisms (e.g., secure storage) negotiated at peering time. Requesters, on the other hand, are expected to take measures to fortify their applications (similarly to public cloud computing) and to configure for each sensitive component the appropriate policies to ensure it is scheduled on security compliant infrastructures only (e.g., private data is processed locally).

4) Fluid topology: Members can join and leave at any time, while spanning across the entire range of infrastructure sizes, from enterprise-grade data centers to IoT and personal devices. Generalizing traditional federation approaches, liquid computing aims at supporting highly dynamic scenarios, with frequent and unexpected (or, in other scenarios, explicitly desired) connections and disconnections. Besides spanning across public and private data centers, as well as edge clusters, the resource continuum possibly encompasses also single devices. This would include IoT, industrial and domestic

[2]https://liqo.io

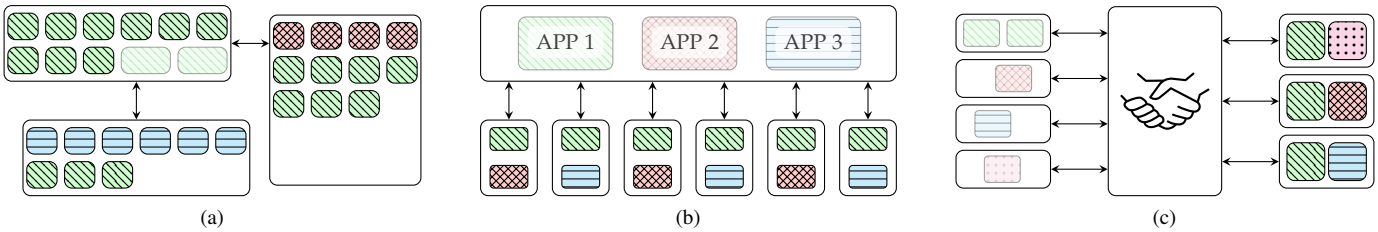

Fig. 1. A graphical representation of the three deployment scenarios fostered by liquid computing: (a) elastic cluster, allowing applications to spill over to federated infrastructures, (b) super cluster, providing an abstraction to control multiple independent infrastructures, and (c) brokering cluster, enabling third parties to match resource demands and offers.

scenarios, all characterized by a multitude of independent appliances typically dedicated to specific tasks (e.g., machine tools control, home automation, monitoring, ...), which could greatly benefit from this paradigm, transparently leveraging the shared resources to offload computations and extend their capabilities [15].

B. Deployment scenarios

Overall, we deem three deployment scenarios to be mostly enabled and fostered by liquid computing (Fig. 1).

1) Elastic cluster: The liquid computing paradigm reduces the fragmentation of scattered clusters thanks to the possibility of transparently leveraging the resources available in other locations, which enables to balance and absorb load spikes (*cloud bursting*). This definitely applies to edge-computing scenarios, whose pervasiveness is typically achieved at the expense of the computational capacity, but is visible also in more traditional cloud contexts in which multiple clusters are active. For example, elastic clusters can be used to support multi-cloud strategies (no single vendor lock-in) and geographically distributed multi-cluster deployments (e.g., due to cost optimization, latency, redundancy or legislative concerns). Resource sharing can involve both on-premise and public data centers, hence deploying latency-sensitive applications close to the final users, while benefiting from the virtually infinite computational capacity featured by larger infrastructures for resource intensive tasks. Thanks to the decentralized approach and the support for peering contracts, resource sharing can occur also when clusters are under the control of different organizations, i.e., they belong to different administrative domains.

2) Super cluster: We are currently witnessing scenarios in which a company owns hundreds, or even thousands, of small clusters, often located at the edge of the network, such as a large telecom operator. In this context, the liquid computing paradigm enables a higher-level, *super cluster* abstraction, representing the single point of entry that can transparently deploy and control applications hosted by the entire infrastructure. Although apparently centralized, each level-2 cluster actually maintains its own local orchestration logic, hence being resilient to network outages and preventing possible synchronization issues arising with relatively high latency WAN links [8]–[10]. In addition, thanks to the super cluster, the administrative burden is greatly reduced, enabling a borderless orchestration that geographically distributes and controls the tasks from a single point of entry, without the necessity of an explicit interaction with the underlying clusters. This scenario facilitates the automatic migration of applications from one cluster to another, which helps when dealing with disaster recovery, infrastructure interventions, scaling, or placement optimization. In addition, it greatly simplifies the replication of jobs across clusters (hence management, monitoring and troubleshooting) as it leverages existing primitives that operate on nodes of the super cluster; for example, an operator can easily replicate the same service on a subset of its edge clusters in order to serve all the end users present in their close vicinity. Finally, this approach can be combined with the elastic cluster for increased dynamism, thus benefiting from the single point of entry for workloads replication, while enabling at the same time the offloading of bursty workloads from the edge to the cloud.

3) Brokering cluster: Complex applications require complex infrastructural setups, accounting for both resiliency and performance. While larger cloud providers could theoretically offer a sufficiently wide catalog of services to satisfy most demands, relying on a single vendor increases lock-in and potentially leads to cost inefficiencies. In this context, we believe that liquid computing could foster the creation of new *Resource and Service Exchange Points* (RXPs), with third party entities behaving as brokers between consumers and providers. Consumers would then only need to peer (both technologically and contractually) with a single RXP to immediately benefit from the entire set of resources (cloud and edge data centers, ...) and ready-to-use services therein offered. This would reduce the complexity and the operational costs especially for smaller companies, lacking the bargaining power of larger enterprises. Resource providers, at the same time, would be encouraged in participating, to easily reach a wider turnout of interested customers. Additionally, even small actors, such as the ones operating at the edge of the network, would be enabled to participate in the edge-cloud market, possibly in a fair competition with far larger giants that may not have enough resources to serve a given geographical area, in a way that looks similar to the energy market in which millions of tiny producers are aggregated by larger buyers.

III. The Liquid Computing Pillars

This section presents the main technical building blocks required to materialize the liquid computing vision, assuming Kubernetes as the orchestration platform leveraged by the underlying clusters. In fact, in our opinion Kubernetes

represents a key enabler for liquid computing, thanks to its capillary diffusion in data centers of any size [19], as well as the support for single devices and IoT computing by means of lightweight distributions, such as *k3s* [20]. To this end, the recent Microsoft's backed *Akri* project [21] goes even further, introducing an abstraction layer to dynamically interconnect to this platform the variety of sensors, controllers and MCU class devices typically present at the very edge of the network. At the same time, Kubernetes can be easily extended through both custom APIs and logic, allowing to transparently integrating liquid-computing related aspects, as well as to semantically enrich the ecosystem and introduce new services that may be shared with peered clusters. Hence, being the underlying platform (conceptually similar to an overarching operating system) the resource continuum is built upon. Still, the key concepts are definitely more general, and can be applied with no particular difference to other orchestrators, such as OpenStack, or even to a mix thereof.

A. Dynamic Discovery and Peering

The first key enabler is the *discovery and peering* function. It fosters the decentralized governance approach typical of a peer-to-peer architecture, preventing the need for central management entities and full administrative control over the entire infrastructure. Additionally, it is responsible for the liquid computing dynamism, allowing for new peering relationships to be established and revoked at any time, compared to the manual coordination required by static federation approaches. In this context, we define *peering* a unidirectional resource and service consumption relationship, with one party (i.e., the consumer) granted the capability to offload tasks and/or consume services in a remote cluster (i.e., the provider), but not vice versa. This allows for maximum flexibility in asymmetric setups, while transparently supporting bidirectional peerings through their combination.

Overall, this module deals with four main tasks. (*i) Discovery*, to identify candidate clusters to peer with, including large enterprise domains, as well as possibly local independent appliances (e.g., IoT devices). (*ii) Authentication*: given the list of feasible candidates obtained during the previous step, optionally filtered through user-configured criteria, it is responsible for the establishment of a secure communication channel with each selected counterpart. Still, resource offloading is not yet possible at this point, being the granted authorizations related to peering establishment steps only. (*iii) Resource negotiation*, involving the exchange of request and offer messages to identify the shortlist of clusters selected for resource offloading. The entire process is policy-driven, with decision modules local to each cluster determining at each step whether to proceed with the negotiation or to abort the process. As a representative example, an offering cluster might implement complex business logic to determine the appropriate prices based on current demands and available resources, accounting for resource brokering and reselling scenarios. Consumers, on the other hand, may filter and rank the received offers by means of appropriate criteria, possibly including compliance with the request constraints, cost, additional attributes, past experience, and more. The negotiation process culminates with the mutual agreement between a consumer and a provider. To this end, we envisage the adoption of smart contracts [22] to formalize the exchange in terms of money and resources, especially in case of inter-administrative domain peerings. (*iv) Peering finalization*: once resource negotiation is completed, the peering relationship needs to be finalized, leading to the exchange of the preparatory parameters required for subsequent computation offloading (e.g., network configurations, as analyzed in Section III-D), as well as the setup of isolation mechanisms and the granting of the suitable permissions in the target cluster.

B. Hierarchical Resource Continuum

Once peering relationships have been established towards one or more targets, the local cluster gains logical access to remote resource slices. Yet, these need to be properly exposed for application offloading through a continuum abstraction, while respecting the limited knowledge propagation and the multi-ownership constraints. Moreover, we deem API transparency to be of utmost importance to foster its widespread adoption, thanks to the introduction of no disruption in well-established deployment and administration practices, as well as the immediate support for existing management solutions.

Being traditional clusters composed of multiple nodes, each one mapping to a physical server, we propose to represent peered clusters through *local, virtual, big nodes*. *Local*, as attached to the consuming cluster; *virtual*, since they abstract a set of remote resources possibly unrelated from the underlying hardware; and *big*, being potentially much larger than classical nodes (in terms of available capabilities), as backed by an entire data center slice. The node concept perfectly complies with the requirement of sharing limited information, hence abstracting peered clusters only in terms of the aggregated resources currently being shared, with no additional details regarding its actual internal configuration. At the same time, it leads to overall better scalability, given the reduced amount of data synced among different clusters. This approach opens up for two possible cluster models. First, *extended clusters*, encompassing a combination of traditional physical nodes (i.e., workers), and virtual ones. This could be suitable for the resource optimization and RXP consumer use-cases, to allow borrowing external computational capacity to overcome local limitations. Second, *virtual clusters*, characterized by the absence of local workers. Combining only virtual nodes, they provide a single point of control abstraction to simplify the deployment of applications on user-defined slices of the underlying infrastructure. Moreover, they represent a key enabler for resource brokering, aggregating the resources offered by multiple providers (each one mapped to a virtual node) for reselling.

The virtual node abstraction leads the underlying orchestration platform (e.g., Kubernetes) to consider the above nodes as valid scheduling targets, hence allowing traditional workloads to be transparently assigned to remote clusters. No differences are perceived by the final users, who simply benefit from the enlarged amount of available resources. This approach brings to a hierarchical representation of the resource continuum.

When a new workload is deployed in the local cluster, the scheduler first selects the optimal node for its execution. Then, if the target is a virtual node, the workload is remapped to the corresponding remote cluster, where it incurs in a second scheduling round to identify the physical server where it will be executed upon. While considering a two-layer scheduling in this example, the approach can easily generalize to multiple levels if needed, depending on the number of virtual node redirections. Hence, allowing scheduling decisions to occur at different abstraction layers, reducing the overall number of feasible candidates to consider at each step and potentially increasing the resulting accuracy. Once more, compliance with standard Kubernetes APIs enables vanilla schedulers to deal out-of-the-box with extended clusters. However, custom scheduling logic might be appropriate in certain scenarios, allowing for further optimizations thanks to the knowledge about the semantics of the peering relationship (e.g., monetary costs, network characteristics, geographical distance, QoS). In both cases, end-users can easily enforce domain-specific constraints through Kubernetes standard high-level policies (i.e., selectors and affinities) to assign workloads to slices of nodes and ensure replicas spreading. Hence, sticking to an intent-driven approach, while requiring no modifications in standard application deployment workflows.

C. Resource and Service Reflection

Each virtual node is responsible for its allocated workloads, whose execution is actually delegated to the remote cluster. Hence, selected control plane information should be present both in the local cluster (required to fulfill the requirement of the virtual node abstraction) and in the remote cluster (enabling the remote control plane to carry out its operations). This introduces the *resource reflection* concept: objects exist both in their *native* form (i.e., in the local cluster), and in their *shadow* form, remotely. Indeed, applications most likely require accessory artifacts for proper execution (e.g., configurations, authorization tokens, network endpoints, etc.), which then need to be reflected in the target cluster. The resource reflection logic enforces the transparent realignment between the two digital twins of the same artifact across the different domains, while ensuring the desired information opacity properties (i.e., omitting or masquerading data that should not be propagated) and resolving possible conflicts which may arise in the remote infrastructure (e.g., naming collisions, different underlying technologies, ...). Overall, it shall support the propagation of both local modifications (e.g., the change in a user configuration) — *outgoing reflection* — and of remote status changes (e.g., an application is being restarted due to a crash), hence allowing for proper inspection — *incoming reflection*. Service endpoints represent one of the most important reflected information, enabling an application running on one cluster to be reachable (hence, consumable) from another cluster. This may require the close coordination of the network fabric (Section III-D) to disambiguate and transparently translate possible overlapped network addresses used in the communication flows.

The clever reflection of the required information is the key to achieve objectives such as *robustness*, enabling clusters to evolve also in case of network disconnections, and *scalability*, reducing the amount of synced data.

D. Network Continuum

According to the virtual nodes approach, different components of the same application may be spread across multiple clusters. Still, the resource continuum, alone, is not sufficient to ensure their correct execution, as the various microservices most likely need to interact among each other.

Orchestration platforms typically implement internal communication by means of private IP addresses, resorting to public ones only for user-facing services. Hence, they are unsuitable for direct (pod-to-pod) cross-cluster interactions and require the introduction of an appropriate *network fabric* responsible for the transparent communication between microservices, no matter where they are executed. Accounting for the decentralized and dynamic approach fostered by liquid computing, with peers possibly joining and leaving at any time, the network fabric cannot rely on ahead-of-time knowledge for its establishment. Indeed, it shall only require the cooperation between the two involved clusters, which negotiate the configuration parameters necessary to set up *(i)* the secured communication channel and *(ii)* the proper mechanisms to guarantee the any-to-any communication across the entire virtual cluster. Being the interconnecting clusters potentially under the control of different administrative domains, it is likely conflicts may arise, e.g., in terms of overlapping IP addresses or underlying networking solutions. The network fabric is expected to transparently handle all these issues, while virtually extending the local cluster network to the entire resource continuum, presenting a unique border-less addressing space.

Supposing a central cluster C peered with n others, we foresee two main network topologies for data plane communications. First, a hub and spoke topology, with n direct interconnections between C and all the leaves. Conceptually simple, this solution requires all traffic between applications residing on peripheral clusters to flow through the central hub, potentially resulting in communication inefficiencies. Still, it may be appropriate when applications do not span across multiple remote clusters (e.g., the same application is replicated in multiple edge clusters), in case either the communication pattern or the underlying network match the star topology, as well as when traffic policies should be enforced from a single point of control. Second, an *opportunistic mesh* topology, providing full connectivity between all clusters hosting applications potentially communicating between one another, to avoid traffic tromboning.

It is worth noting that peripheral clusters may in turn play the role of central clusters for different peering sessions, hence leading to completely dynamic and independent topologies, and potentially overlapped virtual clusters.

E. Storage and Data Continuum

When an application is spread across multiple clusters, stateful workloads require the access to persistent storage locations, which implies a *data continuum* across all the virtual cluster. To this end, we foster the *data gravity* approach

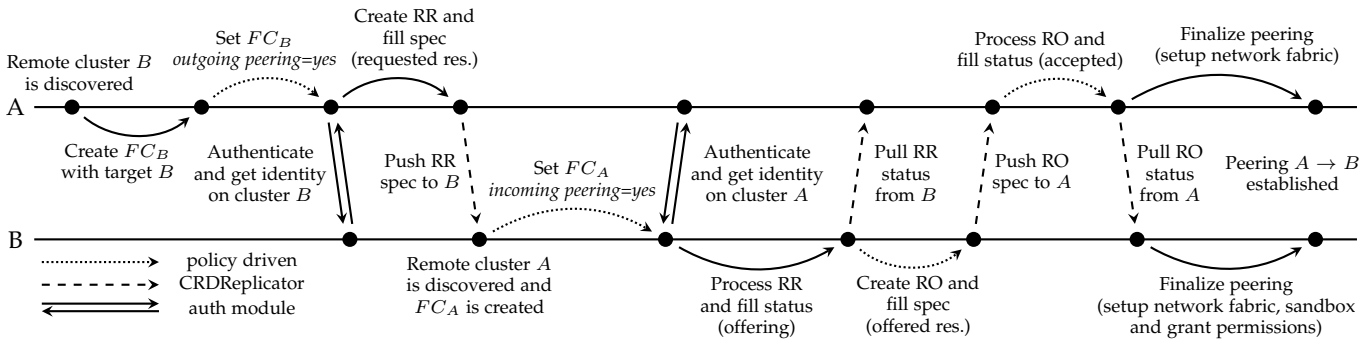

Fig. 2. Schematic representation of the *liqo* discovery and peering procedure. FC: ForeignCluster, RR: ResourceRequest, RO: ResourceOffer.

borrowed from well-established practice in the Big Data world [23]. According to it, data attracts the associated workloads (i.e., introducing additional placement constraints) rather than vice versa, to ensure the best performance in terms of reduced network traffic and latency, as well as to enforce storage locality, which represents a possible strong requirement to comply with law regulations. This paradigm allows also for the extension of traditional in-cluster stateful workloads replication mechanisms (e.g., databases) across the entire resource continuum, transparently achieving increased disaster recovery support. Information replication and synchronization might be further supported if more dynamism is desired, although requiring the exchange of data between potentially distant storage pools; hence, this is mostly suitable only in case of limited amounts of data.

IV. The Liqo Architecture

This section presents *liqo*, an open-source project[3] fostering the liquid computing vision presented above. Acknowledging its wide diffusion and flexibility, *liqo* builds upon and extends Kubernetes to enable dynamic and seamless multi-cluster topologies independently of the underlying infrastructural borders. Overall, *liqo* aims to introduce no modifications in standard Kubernetes APIs for application deployment and well-established management workflows, as well as to support a wide range of common infrastructures, with no constraints in terms of cluster type (i.e., on-premise or hosted by a cloud provider) and networking configurations (i.e., CNIs and IP addresses). In the following, we detail its main architectural characteristics, while building a parallelism with the technical pillars presented in Section III.

A. Discovering and Peering with Remote Clusters

The *liqo* discovery logic is responsible for the identification of possible remote clusters to peer with. Accounting for different scenarios, *liqo* supports (*i*) manual configuration, and standard DNS-based Service Discovery [24], leveraging both (*ii*) conventional Unicast DNS, suitable for large enterprise domains, and (*iii*) Multicast DNS, allowing dynamic on-LAN clustering of independent devices. In all cases, the output is a remote network endpoint that can be later leveraged to start the authentication procedure: this information, along with the desired peering state (i.e., whether it should be established) and possible additional attributes is represented through a *ForeignCluster* Custom Resource (CR).

The *liqo* peering procedure (Fig. 2) starts once a given discovered cluster B is selected as a desired target (i.e., *outgoing peering* flag set in its ForeignCluster CR), either manually or through policies. This procedure is entirely based on a Kubernetes-native logic, which consists in setting the proper resources in Kubernetes, possibly reflected in the other cluster by *liqo*; however, a more traditional protocol-based approach could also be envisioned. The first step involves the authentication module: the originating cluster A generates a new private key locally, then sends a *Certificate Signing Request (CSR)* and a pre-shared token to the remote endpoint. If authentication is granted, the remote module in B proceeds signing the request, assigning A just the bare permissions necessary during the peering establishment procedure, and eventually returning the generated certificate. This approach completely integrates with standard Kubernetes permissions management, and it does not require any common certification authority among peering clusters. Alternative solutions might be adopted in different scenarios, such as to comply with public cloud requirements. In the end, the requesting cluster A shall obtain a valid identity, later used to interact with the remote peer B.

The resource negotiation can now start. First, A creates a new *ResourceRequest* CR locally, to make explicit the desire to request computational resources and/or services to a remote cluster, and configures the content of its *Spec* stanza to convey the desired information. Then, the *CRDReplicator*, which is responsible for the interaction among clusters through the replication of custom resources during the peering establishment, takes action, and it duplicates the CR on the remote cluster. Once the ResourceRequest is received, B automatically discovers cluster A, creates the corresponding *ForeignCluster* representation and, in case it is willing to proceed with the resource negotiation (i.e., the *incoming peering* field is set), it performs the symmetrical authentication procedure. At the same time, the *ResourceRequest* is processed by a custom logic and the outcome, along with possible additional parameters, are back-propagated through an update of its *Status* stanza (eventually pulled by cluster A), which can be used to decline

[3]Source code is available at https://github.com/liqotech/liqo

the peering request. In case of acceptance, B would emit a proper *ResourceOffer* CR to convey the willingness of sharing a given amount of available resources/services, possibly at a given price, and replicate it to the requesting cluster A through the CRDReplicator. *liqo* features resource negotiation based on a customizable amount of available resources, with the support for pluggable decision modules (e.g., for brokering scenarios).

Once a ResourceOffer is accepted, the new peering relationship can be finalized, establishing the inter-cluster network fabric (cf. Section IV-D). Additionally, cluster B grants increased permissions to A, allowing for computation offloading in the target cluster, while configuring at the same time the appropriate isolation mechanisms in terms of network communication, security, resource usage, etc. The established peering relationship is unidirectional, with A being granted the possibility to leverage the resources offered by B, but not vice versa. Still, the reverse procedure can be later started by B, achieving a bidirectional peering.

B. The Virtual Node Abstraction

liqo leverages the *virtual node* concept to masquerade the resources shared by each remote cluster. This solution allows the transparent extension of the local cluster, with the new capabilities seamlessly taken into account by the vanilla Kubernetes scheduler when selecting the best place for the workloads execution. The virtual node abstraction is implemented through an extended version of the *Virtual Kubelet* project [25]. In Kubernetes, the *kubelet* is the primary *node agent*, responsible for registering the node with the control plane and handling the lifecycle of the *pods* (i.e., the minimum scheduling unit, composed of one or many containers sharing the same network namespace) assigned to that node. The virtual kubelet (VK) replaces a traditional kubelet when the controlled entity is not a physical node, allowing to control arbitrary objects through standard Kubernetes APIs. Hence, it enables custom logic to handle the lifecycle of both the node itself and the pods therein hosted.

1) Node lifecycle handling: The first task handled by the VK regards the creation and the management of the *virtual node* abstracting the resources shared by the remote cluster. In particular, it aligns the node status (i.e., whether it is ready, as well as its size in terms of available resources) with respect to the negotiated configuration (i.e., ResourceOffer). Periodic healthiness checks are performed to assess the reachability of the remote cluster, marking the virtual node as *not ready* in case of repeated failures. Upon this event, if disconnections are explicitly foreseen and shall be tolerated (e.g., to account for edge devices in harsh environments), existing workloads are allowed to evolve independently through the remote orchestration logic, with the virtual node no longer considered a valid scheduling target only for new applications. Differently, in other scenarios, user configurations might require standard Kubernetes logic to proceed evicting all pods hosted on the failing cluster and reschedule them in a different location to ensure service continuity.

2) Pod lifecycle handling: Differently from a traditional kubelet, which starts the actual containers on the designated node, the *liqo* VK implementation is conceptually responsible for mapping each operation to a corresponding *twin* pod object in the remote cluster for actual execution, while possibly going through additional indirection levels in brokering scenarios. Still, remote status changes are automatically propagated to the respective local pods, hence allowing for proper monitoring and administrative inspection. Advanced operations, including metrics and logs retrieval, as well as interactive command execution inside remote containers are transparently supported, to comply with standard troubleshooting operations.

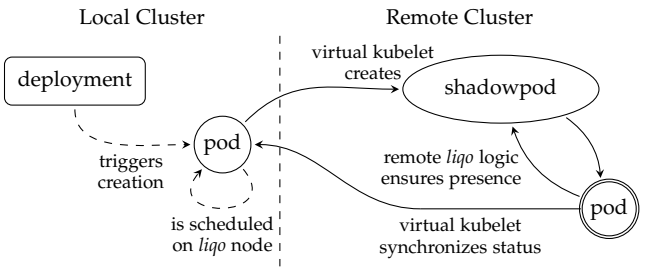

Fig. 3. Schematic representation of the pod offloading workflow. Solid lines refer to *liqo*-related tasks, while dashed ones to standard Kubernetes logic. Double circles indicate the pod in execution (i.e., whose containers are running).

The overall offloading process can be summarized as follows (cf. Fig. 3). First, a user requests the execution of a new pod, either directly or through higher level abstractions (such as *Deployments*), which is then assigned by the Kubernetes scheduler to a virtual node. The corresponding VK instance takes charge of it, creating its twin copy in the remote cluster. However, simply deploying pods remotely would possibly lead to resiliency problems in case of split-brain scenarios (e.g., due to temporary connectivity loss between clusters), causing service disruption if the remote pods were deleted following node failure or eviction. For this reason, *liqo* resorts to the remote creation of a *ShadowPod*, a CR wrapping the pod definition and triggering the remote enforcement logic. Ultimately, it leads to the generation of the corresponding twin pod, while transparently ensuring execution resiliency independently of the connectivity with the originating cluster. In a nutshell, local pod operations (i.e., creations, updates and deletions) are translated to corresponding ones on remote ShadowPods, while automatic remapping is performed by the incoming reflection logic to locally propagate pod status updates in the main cluster when appropriate.

3) Resource and service reflection: The *liqo* VK deals also with the remote propagation and synchronization of the artifacts required for proper execution of the offloaded workloads. The reflection process is enabled by system administrators on a per-*namespace* basis, together with pod offloading (cf. Section IV-C). Yet, specific artifacts (e.g., sensitive secrets) can be manually annotated and excluded. Currently, it supports shadow *ConfigMaps* and *Secrets*, which typically hold application configs, as well as shadow *Services* and *EndpointSlices (epslices)*, to allow for intercommunication between microservices spread across multiple clusters.

In this respect, let consider the example shown in Fig. 4: a given application A is composed of three microservices (i.e., pods), namely P_1, P_2 and P_3, exposed through the respective

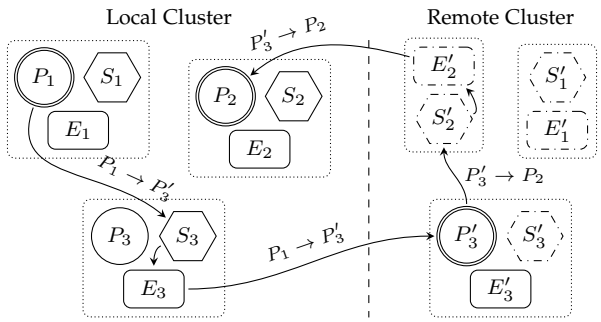

Fig. 4. Graphical representation of the communication patterns between three microservices spread across two different clusters through *liqo*. Dashed polygons represent shadow resources, while double circles indicate that the pod is actually in execution.

service S_i, in turn associated with epslice[4] E_i. Additionally, let assume P_1 and P_2 are executed on local workers, while $P_3 \equiv P'_3$ is offloaded to a remote cluster through a virtual node. Once the *liqo* network fabric is configured to allow inter-cluster pod-to-pod communication (cf. Section IV-D), P_1 can directly contact P'_3 through the corresponding service[5] S_3. As a matter of fact, the local Kubernetes control plane perceives the remote pod as executed locally and, given its (possibly remapped) IP address is present as part of its status thanks to the incoming reflection process, it creates the corresponding epslice entry (i.e., E_3) as usual to allow traffic forwarding. In the opposite scenario (e.g., the remote pod P'_3 willing to communicate with a local one — P_2), the outgoing reflection takes action. First, it creates the shadow copy S'_i of the local services, to enable transparent DNS discovery without requiring IP correspondence. Second, it configures the appropriate epslice entries (possibly remapping the IP addresses, according to the network fabric configuration) to account for the local service endpoints: indeed, these cannot be managed automatically by Kubernetes, as the corresponding pods are not physically present in the remote cluster. In the end, when P'_3 contacts $S'_2 \equiv S_2$, the standard logic forwards the request to one of the IP addresses present in the epslice E'_2, eventually reaching the local pod through the *liqo* network fabric, which performs the appropriate NAT translations if necessary. Multiple replicas of the same microservice spread across different clusters, and backed by the same service S_x, are also handled transparently. Indeed, each pod, no matter where it is located, contributes with a distinct epslice entry, either by the standard control plane or through outgoing reflection, hence becoming eligible during the service load-balancing process (possibly leveraging standard Kubernetes mechanisms to favor traffic locality and reduce inter-cluster communication).

[4]For the sake of dissertation, we assume here a single *epslice* per service, although there may be multiple (mostly for scalability reasons). Indeed, this possibility is leveraged by *liqo* to segregate the reflected entries from the ones referring to local pods and achieve better scalability.

[5]Technically speaking, pods can communicate directly even without exploiting the service abstraction. Yet, the latter is typically leveraged to define a single point of access agnostic from the underlying pods and supporting DNS discovery mechanisms.

C. Workload Scheduling Policies

Each peered remote cluster is associated with a set of labels, key/value pairs describing its main characteristics (e.g., the geographical region, the hosting provider, etc.), configured by its administrators and automatically propagated to the corresponding virtual node. This allows for fine-tuned selection of the cluster(s) each workload shall be executed on, according to its requirements and the resource continuum capabilities. Specifically, *liqo* provides a two-levels selection mechanism. First, administrators can enable remote offloading, along with resource reflection, on a per-*namespace* basis, while possibly selecting for each one a specific subset of remote clusters through their distinguishing labels (e.g., requiring those in a certain country). Advanced configurations are foreseen to tune namespace remapping for collisions handling, as well as possibly preventing workloads scheduling on local nodes or, vice versa, preferring local to remote nodes unless in case of excessive cluster load. Second, additional constraints can be configured at deploy time, to further restrict the eligible targets for each workload based on their requirements (e.g., enforcing front-end components to be hosted close to the end users, while introducing no additional constraints for back-end workloads); hence, fostering the intent-driven approach required by liquid computing. Under the hood, each requirement is mapped to standard Kubernetes mechanisms (i.e., *taints/tolerations* and *affinities*), to comply with established practice and traditional operational procedures.

D. The Liqo Network Fabric

The *liqo* network fabric is in charge of transparently extending the Kubernetes network model across multiple independent clusters, such that offloaded pods can communicate with each other as if they were all executed locally. Traditionally, Kubernetes guarantees that pods on a node can communicate with all pods on any node without NAT translation. *liqo* broadens this requirement, ensuring all pods in a given cluster can communicate with all pods on all remote peered clusters, either with or without NAT translation. Indeed, the transparent support for arbitrary clusters, with completely uncoordinated parameters and components (e.g., CNI) makes impossible to guarantee non-overlapping pod IP address ranges (i.e., `PodCIDR`). This requires the support for IP translation mechanisms, provided that NAT-less communication is preferred whenever address ranges are disjointed. Following industry-standard practice, the clusters interconnection is delegated to secure VPN tunnels, which are dynamically established at the end of the peering process.

The *liqo* network fabric currently implements a *hub and spoke* topology[6] (cf. Section III-D) and it is composed of three main components (Fig. 5). First, the *network manager*, responsible for negotiating the connection parameters (i.e., VPN technology, IP address ranges, etc.) with each remote cluster through the exchange of appropriate CRs. It features also an IP Address Management (IPAM) plugin, which deals with possible

[6]Ongoing work is focusing on the implementation of the opportunistic mesh configuration, to provide direct connectivity between peripheral clusters when appropriate.

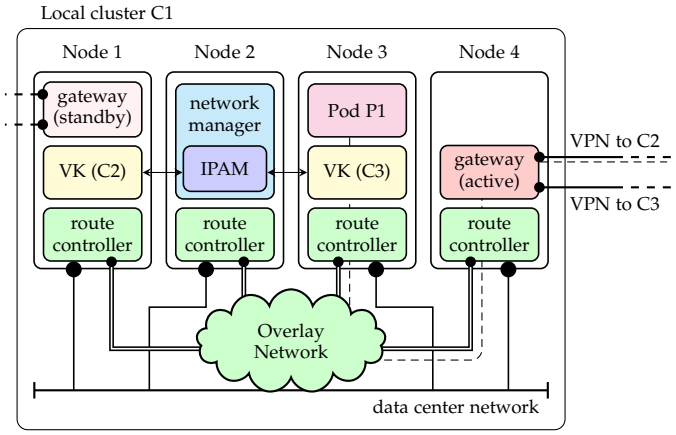

Fig. 5. Main *liqo* components of the network fabric subsystem, including the interconnection between clusters. The dashed line shows the path followed by traffic originating from a pod P1 and directed to one hosted on cluster C2, flowing through the overlay network to reach the gateway pod and eventually entering the VPN tunnel.

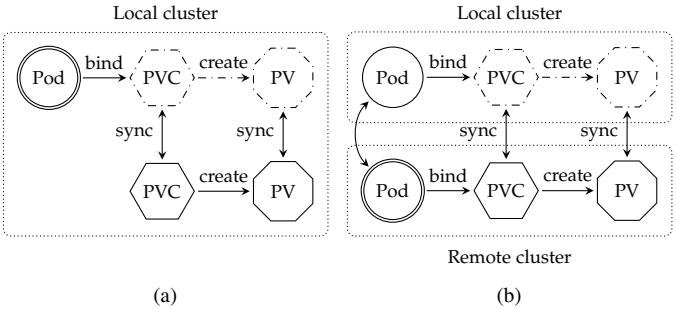

Fig. 6. Graphical representation of the persistent storage provisioning logic, in case the binding pod is scheduled on (a) a physical node or (b) a virtual node. Dashed polygons represent virtual resources, while double circles indicate that the pod is actually in execution.

network conflicts through the definition of high-level NAT rules (enforced by the *gateway*), while also exposing an interface consumed by the *liqo* VK reflection logic to handle IP addresses remapping. This can occur in case of overlapping `PodCIDRs` between clusters, which are transparently managed through a 1:1 translation rule to a second equivalent address range negotiated at peering time. Additionally, ad-hoc remapping to an external free pool of IP addresses is also foreseen to support the communication between two arbitrary pods (through the respective services, hence *epslices*) hosted by two different peripheral clusters in case of indirect address conflicts.

The second component is the *gateway*, in charge of the setup of the VPN tunnels towards remote clusters, based on the negotiated parameters. It implements a generic southbound interface to allow for multiple underlying drivers, although *liqo* currently supports only the *WireGuard* [26] plugin, a modern VPN solution with state-of-the-art cryptography. Additionally, it appropriately populates the routing table, as well as translates and installs, leveraging *iptables*, the different NAT rules requested by the network manager. Although this component is executed in the *host network*, as dealing with the node networking stack, it relies on a separate network namespace and policy routing to ensure isolation and prevent conflicts with the Kubernetes CNI plugin. The gateway supports *active/standby* high-availability, to ensure minimum downtime in case the main replica is restarted.

Finally, the third element is the *route controller*, a *DaemonSet* (i.e., a component executed on all physical nodes of the cluster) that is responsible for configuring the appropriate routing entries, and possibly an overlay network, to forward all traffic from local pods/nodes and directed to remote clusters through the gateway (and thus the VPN tunnel). Once more, the high-level control logic leverages an abstract southbound interface, to allow for multiple underlying technologies. Specifically, *liqo* currently supports both a direct routing setup, hence leveraging the native infrastructure whenever possible, and a *VXLAN*-based overlay network, for scenarios incompatible with the previous approach.

E. The Liqo Storage Fabric

Along with the support of stateless workloads, *liqo* transparently enables the offloading of stateful tasks through a transparent inter-cluster storage continuum. This feature is enabled by the storage fabric subsystem, which tackles the problem through two different techniques. First, adopting the *data gravity approach* detailed in Section III-E: whenever a workload is required to access an already existing pool of storage, a set of automatic policies forces its execution in the appropriate cluster. Second, deferring storage binding until its first consumer is assigned to a given cluster, thus ensuring new storage pools are created in the exact location where their associated workloads have been scheduled to.

Albeit simple, these approaches extend standard Kubernetes practice to the entire resource continuum, as well as they fulfill most common use-cases. Let consider first a high availability and disaster recovery scenario, with a database instance that needs to be replicated among different clusters. Multiple member replicas can be spawned leveraging traditional in-cluster mechanisms (typically relying on the *StatefulSet* abstraction), while configuring at the same time the appropriate intent-driven policies to enforce spreading across different virtual nodes (i.e., clusters). Upon scheduling, a new storage pool is created in the appropriate location, and associated with each replica, which will continue to be attracted by that virtual node even following subsequent restarts. As a second representative example, let consider an existing storage pool, attached to either a local or a remote cluster, which contains the data to be processed by a batch job. Upon creation, the job is automatically constrained to be executed by the (virtual) node owning the corresponding piece of storage. Hence, ensuring it can be accessed directly, without the need for expensive copy operations and enforcing at the same time data locality, which might be required by law regulations or corporate policies.

Kubernetes leverages the *PersistentVolumeClaim* (PVC) abstraction to represent a request for storage by a user, which eventually leads to the provisioning of a *PersistentVolume* (PV) (i.e., the actual piece of storage a pod can bind to), either manually or through a *StorageClass*. In this context, *liqo* implements a *virtual* storage class, which embeds the logic to create the appropriate storage pools on the different clusters. Whenever a new PVC associated with the virtual storage class

is created, and its consumer is bound to a (possibly virtual) node, the *liqo* logic goes into action (cf. Fig. 6). If the target is a physical node, PVC operations are then remapped to a second one, associated with the corresponding *real* storage class, to transparently provision the requested volume. Differently, in case of virtual nodes, the reflection logic is responsible for creating the remote shadow PVC, remapped to the negotiated storage class, and synchronizing the PV information, to allow pod binding. Finally, locality constraints are automatically embedded within the reflected PVs, to force each workload to be scheduled only on the clusters where the associated storage pools are available.

V. Implementation Details

We have implemented *liqo* in about 30 000 lines of Go. According to standard practice in Kubernetes, we leveraged Custom Resource Definitions (CRDs) to describe the user-facing APIs for *liqo* configuration and its internal status. Overall, we defined more than ten new APIs, describing the remote discovered clusters, along with the desired peering status, dealing with resource and network parameters negotiation, as well as expressing namespace offloading policies and supporting resilient remote pod execution. The business logic is implemented in accordance with the standard Kubernetes *operators* pattern, with each controller responsible for enforcing the observed status in the cluster to match the desired one expressed by means of the corresponding resource (i.e., CR). This paradigm guarantees separation of concerns between each component (i.e., each controller deals with a single resource, and it is responsible for a precise subset of operations), while the control loop-driven approach ensures that failures are automatically corrected, eventually reaching the desired state. Feedback is returned to the administrators according to standard approaches, updating the *status* stanza of the corresponding resource and through Kubernetes events. Each operation performed by the controllers (e.g., creation or update of existing resources) is idempotent, ensuring that temporary errors and component restarts are handled correctly, without undesired side effects.

The overall *liqo* code-base is subdivided in multiple cooperating components, each one packaged as a separate Docker container and executed by the hosting Kubernetes cluster. Besides the network fabric detailed in Section IV-D, *liqo* includes the following four components.

liqo-controller-manager: it groups together the main operators dealing with *liqo* resources. We leveraged the controller runtime project [27], an abstraction built on top of the Kubernetes client to streamline the implementation of operators and efficiently use shared object caches to reduce the interactions with the API server.

liqo-virtual-kubelet: executed in one replica for each remote cluster, ensuring isolation and segregating the different authentication tokens, it is responsible for the lifecycle of the virtual node and of the pods therein hosted, as well as for resource and service reflection (cf. Section IV-B). Being the key component responsible for the computation offloading performance, it is implemented leveraging lower-level concepts such as *informers* and *working queues* [28] (i.e., the operators building blocks, as also done by core Kubernetes components) to increase the control and reduce possible penalties inside the offloading *fast path*, regardless of the number of objects processed in parallel. Each resource type (e.g., pods, services, epslices, ...) is associated with a custom reflection routine, accounting for parameters remapping and reduced information sharing. Finally, smart caching mechanisms limit the interactions with the Kubernetes API server and with other *liqo* components (e.g., to handle pods and epslices address translation).

liqo-crd-replicator: component responsible for the interaction between peering clusters, enabling resource negotiation and network setup procedures through the exchange of CRs. It leverages a custom manager module, starting and stopping the resource synchronization logic towards each remote cluster based on the current peering phase and the overall configuration. Resource synchronization is implemented through a generic routine that enters in action whenever either the local or the remote version of a marked resource is modified (as detected by Kubernetes *informers*), and realigns the two digital twins, with the local copy being the source of truth for the *spec* stanza, and the remote one for the *status* stanza. Hence, it transparently implements back and forth communication protocols through Kubernetes CRs.

liqo-webhook: a mutating webhook enabling the appropriate subset of pods to be potentially scheduled on virtual nodes, based on the configured high-level policies. Specifically, this is performed enriching the pods specification with the appropriate *toleration* for the virtual nodes *taint*, as well as introducing additional node *affinity* constraints.

The deployment of the *liqo* components is managed through a Helm chart, as per standard practice. Furthermore, *liqo* features also a CLI tool (*liqoctl*) that streamlines its configuration, automatically retrieving the appropriate parameters depending on the underlying environment (e.g., cloud provider, network setup). Additionally, it simplifies the manual definition of peering candidates and the selection of local namespaces for offloading to remote clusters, along with the specification of possibly complex policies. Finally, *liqo* is compatible with on-premise Kubernetes clusters (both vanilla and OpenShift-based), managed clusters hosted on major cloud providers, including Amazon EKS, Microsoft AKS and Google GCP platforms, and lightweight distributions, such as *k3s*. Further details about the full compatibility matrix are available in the official online documentation.

VI. Experimental Evaluation

This section presents the experimental evaluation of the most recent version of *liqo* at the time of writing (v0.4.0) as an insight of the potential performance and scalability properties of the liquid computing paradigm, taking into account the more limited scope of the current implementation.

A. Peering Establishment

Given the *fluid topology* (Section II-A) characteristic of liquid computing, with a potential huge number of (short-living) peers, this test assesses the scalability of the peering establishment

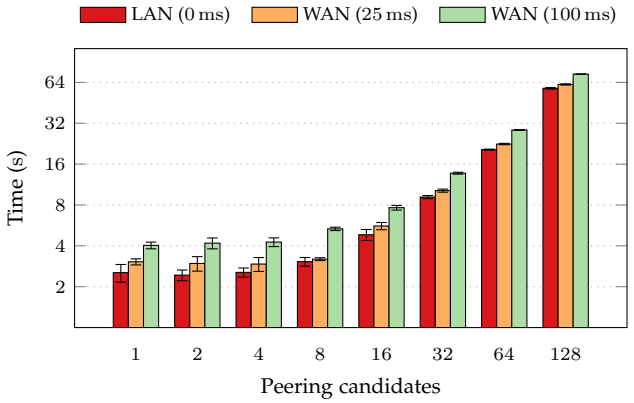

Fig. 7. Peering establishment performance varying the number of peripheral clusters, and the latency with respect to the hub.

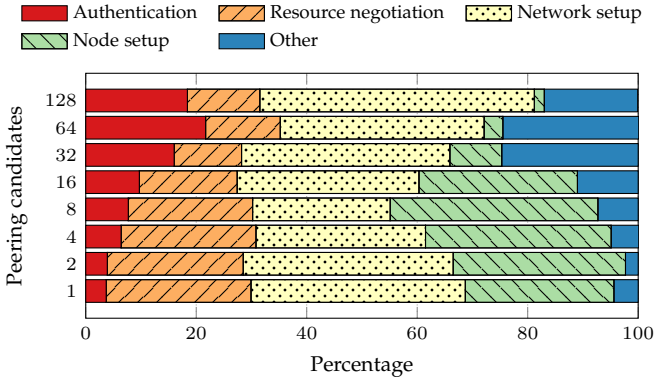

Fig. 8. Peering establishment performance varying the number of peripheral clusters, break down by component (on-LAN scenario).

process, to evaluate the time elapsing from the discovery of a new peering candidate to the creation of the associated virtual node, while varying the number of target clusters. The testbed consists of n Kubernetes clusters in the *super cluster* configuration, with a central entity (*hub* cluster) establishing uni-directional peerings towards all peripheral clusters. To simplify the setup and tear down of the entire testbed, as well as guaranteeing the replicability of the experiments, each cluster is implemented by a *k3s* (v1.21.3-k3s1) instance executed within a Docker container, all together hosted by a single Kubernetes cluster.[7] Each *k3s* cluster is by no means characterized by reduced functionality compared to a bare-server installation, while *k3s* itself likely represents a privileged distribution for edge-oriented scenarios, thanks to its reduced demands in terms of computing resources. To further reduce possible interference between the different instances, we leveraged the more performance-oriented *etcd* database (instead of the *k3s* default, *SQLite*) and mounted its directory to a RAM-backed file-system, hence, preventing concurrent disk access bottlenecks when increasing the number of clusters hosted by the same physical worker. Similarly, relevant Docker images are retrieved in advance, to prevent their download from the Internet during the actual tests. All measurements have been performed through a custom tool executed on the hub cluster, which is responsible for identifying the peripheral clusters and starting the peering process (i.e., creating the corresponding *ForeignCluster* resource), while monitoring the time required to complete each of the different peering phases. The complete artifacts required to replicate the setup and perform the measurements are available on GitHub.[8]

Fig. 7 presents the outcome of the benchmark, displaying the time elapsed from the beginning of the peering process up to its completion (i.e., when all the virtual nodes are ready for application offloading), for a number of peripheral clusters ranging from 1 to 128. To assess the impact of the distance between the hub and the peering candidates, we consider three different scenarios:[9] *(i)* negligible latency, with all clusters located in the same LAN; *(ii)* 25 ms RTT latency, compatible with different sites spread across Europe; *(iii)* 100 ms RTT latency, accounting for intercontinental links. All measurements are repeated ten times, with the error bars representing the resulting standard deviation. Results show that the total time increases mostly linearly with the number of parallel peering candidates, while being characterized by a constant lower bound when dealing with less than ten clusters. The overall trend is consistent regardless of the underlying network latency, with the European scenario introducing a $10\,\% - 20\,\%$ overhead and the intercontinental one being associated with a relatively higher burden (in both cases especially when performing few peering establishments in parallel).

The breakdown of the previous numbers according to the most important steps and averaged across all peering sessions performed in parallel (subject to the degree of parallelism enabled by the different *liqo* components), is presented in Fig. 8; results remain mostly constant during the entire evaluation, with authentication, resource negotiation and network setup being the most demanding steps in all the considered scenarios. Indeed, the first requires computationally expensive cryptographic operations, while the others involve parameter negotiations between the two peering clusters, which become even more prominent in case of the intercontinental scenario due to the increased network latency. The node setup, which is started in parallel to the network setup, impacts primarily in case of few peering candidates, with its total time in case of large number of peerings being marginally larger than the network setup. In this case, on the other hand, the *other* phase, which represents the time required for information propagation downstream the peering pipeline, gains relevance because the subsequent steps are busy processing different candidates. Overall, these results confirm the scalability of the *liqo* peering process, which required way less than one second for each target cluster in the most demanding scenario. Finally, overall numbers may be further reduced by tuning the parallelism of the *liqo* logic, although at the expense of an increased resource consumption which, currently represents a very limited cost (more details in Section VI-E).

B. Application Offloading

The second benchmark analyzes the capability to start a huge burst of pods, which may be impacted by the hierarchical

[7]The hosting Kubernetes cluster was composed of ten worker nodes, each characterized by 16 virtual cores and 64 GB of RAM.

[8]https://github.com/liqotech/liqo-benchmarks/

[9]Additional latency is emulated on the hub cluster through *netem*.

scheduling capabilities of *liqo*. We compared the pod startup time in vanilla Kubernetes, *liqo*, and alternative open-source solutions such as *Admiralty* (v0.14.1) and *tensile-kube* (v0.1.1-24-g2bd91c2). Both projects also leverage the VK abstraction, although adopting different approaches under the hood (cf. Section VII). Indeed, *Admiralty* focuses on a custom scheduling logic, while *tensile-kube* adopts an offloading approach similar to *liqo*, but it implements no remote resiliency mechanisms to prevent split brain scenarios. It is worth mentioning that neither solution includes an automatic peering mechanism (as far as their open-source version is concerned), and thus have not been considered in the benchmark in Section VI-A.

We leveraged a testbed composed of two *k3s* clusters (executed within a container as in the previous case), one playing the role of the *resource provider*, and the other of the *consumer*, hence sticking to the *elastic cluster* scenario.[10] For scalability reasons, worker nodes (i.e., those actually executing the offloaded applications) are represented by *kubemark hollow nodes* [29], which are backed by a component, named *hollow kubelet*, executed in its own container, and that pretends to be an ordinary kubelet, but it does not start any container it is assigned to, it just lies it does. This allows to start a massive number of (fake) containers, even tens of thousands (i.e., comparable with the maximum number of pods supported by Kubernetes [30], $\approx 150\,\mathrm{k}$), with limited resource demands given that containers are not actually running. This does not invalidate the results, given the reduced startup time affects all solutions equally; vice versa, it better highlights the possible overheads introduced by the offloading process, compared to vanilla Kubernetes. All hollow kubelets (100 in our setup) connect to the provider, registering as an additional node. We adopted the official hollow kubelet code base available upstream (v1.21.4), with a simple modification to assign pods an IP from the correct `PodCIDR`, instead of a fake one, for increased realism. All measurements have been performed through a custom tool, which creates the appropriate deployments and waits for the corresponding pods to be generated, possibly offloaded, and become ready. We initially executed it directly on the resource provider cluster to determine the vanilla Kubernetes baseline, and then on the consumer (each time peered through a different technology with the provider) to assess the offloading performance.

Fig. 9 presents the outcome of the evaluation, depicting the time elapsed from the creation of a deployment up to the instant all generated pods are effectively ready, for a number of pods varying between 10 and 10 000. All measurements are repeated ten times, and the error bars represent the resulting standard deviation. The outcome of the benchmark is twofold. On the one hand, both *liqo* and *tensile-kube* displayed excellent performance, introducing practically no overhead compared to vanilla Kubernetes. Still, *liqo* supports additional mechanisms to ensure application reliability even in case of split brain scenario, which *tensile-kube* does not. Differently, the scheduling-driven approach adopted by *Admiralty* turned out to be associated with much worse performance, introducing unbearable overhead when offloading a high number of pods at the same time.

[10]The testbed was hosted by a Kubernetes cluster composed of six worker nodes, totally encompassing 332 virtual cores and 2 TB of RAM.

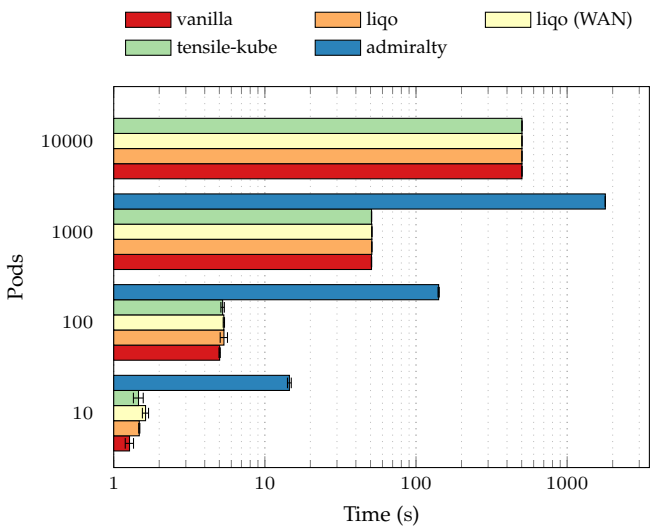

Fig. 9. Application offloading performance comparison, varying the number of pods to be started. The *Admiralty* result is not included for the 10 k pods case, given it is largely out of scale.

As for *liqo*, we additionally evaluated the performance in case consumer and provider are interconnected by a high-latency (100 ms) WAN: no relevant difference emerges compared to the on-LAN scenario, with a slight overhead visible only when offloading 10 pods in parallel (approx. 100 ms). Finally, we varied the number of deployments originating the target set of pods, accounting for multiple use-cases (i.e., ranging from a massively replicated monolithic workload, to a complex application composed of a hundred microservices, each with a significant number of replicas). No significant outcome emerged in addition to the previous considerations, with the difference between vanilla Kubernetes and *liqo* being always smaller than the error bands. Hence, we have omitted these results from Fig. 9 for the sake of conciseness.

C. Service Exposition

This first test evaluates the time required by the *liqo* reflection logic to replicate a service and all the associated *epslices* to a remote cluster, hence making them available for consumption by remote applications, but without including the time needed for the vanilla Kubernetes data plane configuration (e.g., kube-proxy). This highlights the time required to propagate a new service (or a new running endpoint) across the control plane of the virtual cluster as well as the scalability of the solution. We leveraged a testbed similar to the previous one, characterized by two *k3s* clusters and a set of hollow nodes to host the fake containers. However, we considered the symmetric scenario, with a varying number of pods started locally and, once ready, exposed through a single Kubernetes service. A custom tool is responsible for measuring the time required to fill all epslice entries, both on the local cluster (i.e., by the vanilla Kubernetes logic) and on the remote one (i.e., through the *liqo* reflection logic).

The outcome of the benchmark is depicted in Fig. 10, which shows the ten-runs average of the time elapsed from the creation of a service targeting the given number of pods to the

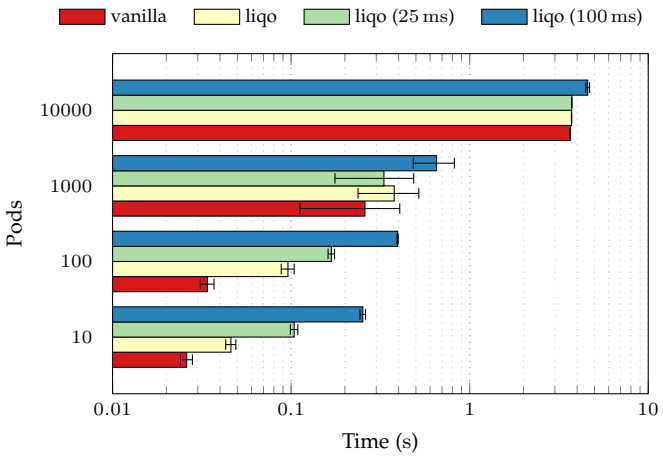

Fig. 10. Service exposition performance comparison, varying the number of endpoint pods and the inter-cluster latency.

complete creation of the corresponding epslices (marked as *Ready*). The graph confirms the limited performance overhead introduced by the *liqo* reflection logic compared to vanilla Kubernetes, accounting for a few milliseconds only even in the most demanding scenario. Given the overall short times required to complete the process, the effect of the underlying network latency, both considering the European ($25\,\mathrm{ms}$) and the intercontinental ($100\,\mathrm{ms}$) scenarios, becomes relevant. Yet, in absolute terms, the overhead is definitely close to the network latency itself, which is unavoidable.

To further characterize the service propagation overhead, we additionally measured the time elapsed from the creation of a service to the instant it is fully reachable; therefore, including the reflection logic, the configuration of *iptables* rules by vanilla kube-proxy, and the network fabric data plane contribution (e.g., packets traversing the *WireGuard* tunnel). In this scenario, we leveraged a single *nginx* pod as service endpoint, with a custom tool executed in both clusters and continuously probing the service through TCP SYN segments, until the corresponding acknowledgement is received; hence, confirming the service is reachable. Across ten runs, the local service (i.e., where the back-end pod is running) became accessible in $0.091 \pm 0.012\,\mathrm{s}$. As for the remote cluster, the *liqo*-reflected service turned reachable in $0.100 \pm 0.008\,\mathrm{s}$ (in case of negligible inter-cluster latency), and $0.218 \pm 0.032\,\mathrm{s}$ in the WAN ($100\,\mathrm{ms}$) scenario. Overall, showing once more limited overhead compared to vanilla Kubernetes, despite the increased functionality.

Focusing finally on network throughput, the data plane handling the actual communication between any two pods hosted by different clusters relies on standard VPN technologies (i.e., WireGuard) and inherits their performance, as well as those of the underlying network.

D. Stateful workloads

This benchmark assesses the performance of the *liqo* storage fabric subsystem, concerning both the creation of new PVs and the binding of a pod to an already existing volume. We adopted a testbed composed of two *k3s* clusters, complemented by a custom tool responsible for the creation of a *StatefulSet* (i.e., the Kubernetes abstraction representing a set of pods with consistent identities, each characterized by one or more volume claims) and the measurement of the pod startup time (including volume creation and binding). Concerning persistent volumes management, we evaluated the usage of a vanilla storage class (i.e., the one included by default with *k3s*), as well as of the *liqo*-provided one, when pods are hosted by either the local or the remote cluster.

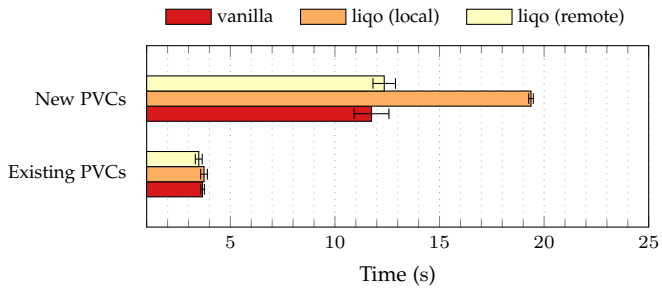

Fig. 11. Stateful workloads startup performance comparison, both when the underlying storage needs to be created and when it already exists.

Fig. 11 presents the ten-runs average of the startup time in case of a *StatefulSet* originating five replicas,[11] hence mimicking a high-availability database setup. First, we analyzed the initial deployment of the application (*New PVCs* in figure), which includes the creation of the PVCs, that of the underlying PVs, and the startup of the pods themselves. Results associated with the vanilla and *liqo* (remote) scenarios are aligned, while the setup of volumes on the local cluster through *liqo* turned out to be slower. This limitation traces back to an external library we leveraged, which adopts by default a rather long polling period to detect the creation of the actual PVs. While better performance could be obtained fine-tuning that component, it is worth mentioning that the creation of PVCs in cloud-provider environments is typically much slower (on the order of minutes), as well as this operation is expected to be quite infrequent. Differently, no relevant difference emerged when binding the pods to already existing volumes (i.e., *Existing PVCs*), which instead happens whenever one or more replicas are restarted.

E. Liqo Resources Characterization

The last test characterizes the *liqo* resource demands, in terms of CPU and RAM required for the control plane execution, as well as the network traffic generated by *liqo* towards the remote Kubernetes API servers during the different operational phases (e.g., peering, resource offloading, etc.). Overall, the testbed is similar to the one adopted in Section VI-A, and composed of eleven *k3s* clusters, one playing the role of the hub and ten behaving as peering candidates. CPU and RAM consumption is retrieved every second on each cluster through the APIs exposed by the *containerd* container runtime, while network traffic is measured on the hub by means of a custom *libpcap*-based program.

[11]The *StatefulSet* was configured to start all replicas in parallel, rather than sequentially, to better stress the storage subsystem.

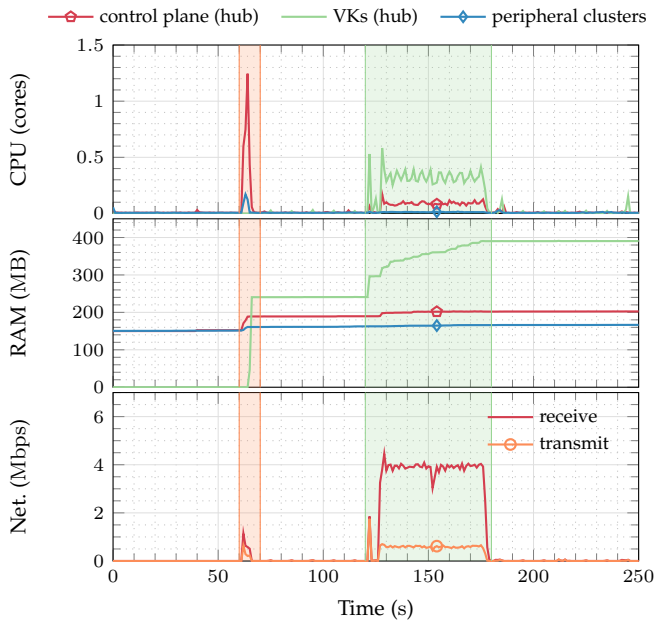

Fig. 12. *liqo* resource demands at rest, peering with 10 peripheral clusters (orange shaded area — 60 s – 70 s) and offloading 1 k pods (green shaded area — 120 s – 180 s). Network metrics refer to the traffic towards remote API servers, measured on the hub cluster (i.e., hosting the VKs).

Fig. 12 presents the outcome of the measurements, subdivided into the *liqo* control plane of the hub cluster (i.e., all the *liqo* components excluding the VKs), the sum of the ten VKs hosted by the hub cluster, and the *liqo* control plane of the peripheral clusters (no VKs are present in this case, since peerings are unidirectional). As for the latter, CPU and RAM metrics are averaged (differences are not significant), hence showing the average requirements for a single cluster. Overall, we consider five different usage phases (highlighted with different colors in Fig. 12): (*i*) At rest and with no active peering (0 s – 60 s): in this context, *liqo* requires approx. 150 MB of RAM on each cluster, with almost no CPU usage and zero network traffic. (*ii*) While peering with ten peripheral clusters in parallel (60 s – 70 s) to assess the processing cost of such operation: the local control plane is characterized by a short CPU spike during the process and a few MB increase in memory consumption, while the ten VKs, started in parallel in the hub cluster, account for approximately 250 MB of additional RAM in total. The exchanged network traffic is negligible. (*iii*) At rest and with the virtual nodes ready (70 s – 120 s): no CPU and network resources are required by *liqo* to maintain the peerings active, while the memory occupancy remains stable compared to the previous phase. (*iv*) Offloading 1 k pods to simulate high churn rates (e.g., a large number of pods is started or changes its state) (120 s – 180 s): as for CPU usage, the ten VKs (as a whole) required a third of a CPU core, while both local and remote control plane demands remained definitely low. VKs RAM usage increased as well, since memory consumption is directly related with the number of pods, according to the standard Kubernetes operators implementation (i.e., watched resources are cached by informers). The information synchronization between the different clusters resulted at the same time in an additional network traffic, although it never exceeded 4 Mbps in total at least in the significant experiment depicted in Fig. 12. (*v*) At rest, with the active peerings and the offloaded pods running on the remote clusters (180 s – 250 s): none of the considered metrics displayed variability, confirming the negligible demands in absence of transient periods.

Concerning the offloading phase, we repeated the entire evaluation for different numbers of pods, while keeping constant the other parameters. Overall, we observed similar CPU usage and network traffic, although for time frames proportional to the number of pods (e.g., ≈ 30 s with 500 pods), according to Kubernetes deployment pacing. As for the VKs RAM usage, the theoretical linear correlation was not completely reflected in the actual measurements, due to the Go garbage collector behavior.[12] In our tests, each offloaded pod accounted for 150 kB – 300 kB of additional RAM, with the upper range associated with lower numbers of overall pods. In addition, different experiments done with different infrastructures (e.g., node characteristics) achieved very similar patterns, with approximately the same amount of total CPU consumed and traffic exchanged, although constrained in shorter (with more powerful nodes) or longer (with less powerful nodes) intervals. Furthermore, the above values proved to be slightly lower compared to the ones generated by ten standard kubelets controlling vanilla Kubernetes worker nodes in the same scenario.

Finally, perhaps the only metric which might deserve attention in the context of constrained devices is memory usage (≈ 160 MB on each peripheral clusters). Yet, it is worth mentioning that even lightweight Kubernetes distributions do require non-negligible amounts of RAM (e.g., *k3s* recommends at least 1 GB [31], which is also confirmed by the measurements in [32]), as well as, at the time of writing, *liqo* could be further optimized to reduce its demands.

F. Additional Considerations

Our experimental evaluation demonstrates the extremely limited overhead introduced by *liqo* in terms of additional resource demands and with respect to vanilla Kubernetes. Hence, justifying the sustainability of the resource continuum abstraction, with its distinguishing characteristics, and of the building blocks enabling the scenarios detailed in Section II-B. In the following, we first present a production environment benefiting from the elastic cluster scenario enabled by *liqo*, and then discuss its potential to overcome Kubernetes scalability limitations, as well as service reliability aspects.

1) Real Liqo deployment: job bursting for online exams: Politecnico di Torino recently hosted different computer science exam sessions on CrownLabs [33], an open-source project started during the coronavirus pandemic onset to deliver remote computing laboratories, later extended and integrated with the official exams platform of our university. It allows each candidate to access her own dedicated remote application instance (started on-demand and executed as a container) of the desired environment (e.g., a full-fledged IDE such as PyCharm) from a standard web browser, while providing automatic project

[12] We leveraged the default GC settings during all tests.

delivery and enforcing the appropriate restrictions to prevent cheating. However, the Kubernetes cluster typically hosting the CrownLabs user instances was not capable enough to sustain the foreseen number of students ($500 - 600$ per round, multiple rounds per day) with the desired resources (i.e., 1 CPU reserved to each instance). The elastic cluster scenario enabled by *liqo* allows part of the instances to be transparently offloaded to a secondary cluster located in a different campus area, while requiring no modifications to the CrownLabs control plane. From extensive monitoring data, no significant performance differences emerged between the creation of local and remote instances, with the *liqo* network fabric handling an average cross-cluster user traffic of $100\,\mathrm{Mbps}$ and the storage continuum ensuring data persistence even in case of instance restarts. Agility proved to be one of the most distinguishing features, allowing to reserve the remote resources only during the actual exam sessions, and immediately releasing them for different purposes at the end. Conversely, standard practice would require to either allocate additional servers to physically extend the cluster, or migrate the entire service to a larger infrastructure, with both alternatives introducing high organizational and operational overheads.

2) Scalability: Liqo vs. Kubernetes: According to the official documentation [30], Kubernetes is currently characterized by scalability upper bounds both in terms of supported nodes (≈ 5000) and pods ($\approx 150\,000$). In the context of *liqo*, these limitations mainly relate to the super cluster scenario, with a single entry point potentially controlling a large number of e.g., edge clusters. Indeed, thanks to the combination of the virtual node abstraction and the remote enforcement logic, *liqo* allows to transparently deal with cluster control planes spread geographically, supporting also scenarios with unstable network connectivity and high latency, which is not possible with vanilla Kubernetes, whose control plane is fully centralized. At the same time, *liqo* has the potential to overcome the Kubernetes node limitations, given that it abstracts an entire cluster with a single node (preventing the propagation of most remote status changes) and it supports hierarchical topologies characterized by multiple indirection levels. For instance, a large company operating thousands of edge clusters (e.g., telco edge; energy smart grids; branch offices, etc.) might leverage regional clusters as intermediate aggregation points, in turn controlled from a single national data center. Hence, preventing to exceed node limitations in any single cluster, while dealing with much higher numbers as a whole. Focusing on pod offloading, as discussed in Section IV-B, *liqo* currently favors full Kubernetes API transparency, which requires all pods to be *virtually* present in the super cluster (i.e., where they are originally created through higher-level abstractions) and accurate status synchronization. Although consuming no local resources, they are nonetheless present in etcd, partially counting towards the Kubernetes limits. This is inherent in the VK approach: future work could focus on additional offloading solutions that are more suitable for high cardinality scenarios, trading off full API compliance with increased scalability.

3) Application reliability: As discussed in Section IV-B, the *liqo* VK leverages periodic healthiness checks to evaluate the reachability of the remote cluster, mapping the outcome to the node readiness property. This perfectly resembles the behavior of vanilla Kubernetes nodes, allowing at the same time the pods therein hosted to obey to standard eviction policies to enforce application reliability. In particular, two main parameters control the entire process, indirectly determining the maximum time frame between a remote cluster turning unreachable and the hosted pods being rescheduled in a different location. First, the node lease duration (default: $40\,\mathrm{s}$): the VK periodically renews a *lease* (i.e., updates an appropriate resource) to confirm it is operating correctly. In case the check fails, the VK stops doing so and the lease expires after that period, causing Kubernetes to mark the node as unreachable. Second, pods toleration for *not ready* and *unreachable* nodes (default: $300\,\mathrm{s}$), that is the maximum interval the pod is allowed to remain bound to a problematic node, before being evicted and scheduled to a different one. In other words, the entire process requires by default at most $340\,\mathrm{s}$. Depending on the specific scenario, different settings may be more appropriate: lowering both values, and in particular the toleration period (which might be even set to $0\,\mathrm{s}$), allows for faster reactions, at the cost of potentially higher churn rates in case of temporary connectivity issues. Differently, drastically higher toleration settings might be better suited in harsh environments, to explicitly account for temporary network partitioning while letting existing workloads to evolve independently, thanks to the remote enforcement logic, guaranteed by the control plane running in the remote cluster. Still, tolerations are set per pod, allowing for fine-grained control and application specific settings, regardless of the specific target cluster.

In other words, upon cluster disconnection, the orchestration logic in the main cluster can be either configured to immediately re-spawn, in another location, all the services that are no longer available for the sake of service continuity, or to leave services where they are. This accounts for either the case in which services should be always available to the users of the main cluster (hence, are re-spawn elsewhere), or the services are intended for local users of the disconnected cluster, which presumably are still able to reach their local infrastructure, even in case of unreachability of the main cluster. In the latter case, the remote control plane features a dedicated enforcement logic that ensures improved service resiliency, guaranteeing that the potential failure of a local node leads to no service disruption thanks to automatic pod rescheduling, regardless of the connectivity with the main cluster. This is different compared to a vanilla Kubernetes cluster encompassing nodes spread geographically, as connectivity loss in a given area would isolate that group of nodes from the central control plane, lacking the possibility for any evolution of their state.

VII. Related Work

The effort towards a transparent resource continuum dates back to the eighties and the concept of distributed operating systems, aiming to abstract a set of independent, autonomous and communicating CPUs that appear to users as a single computer [34], [35]. At the same time, much work focused on a common substratum for applications execution. First, by means of high-level programming languages (e.g., Java [36])

to achieve architecture neutrality, and later through containerization, providing a lightweight answer to packaging and distributing interoperable applications [1]. This paved the way for container orchestration platforms such as Kubernetes, abstracting the resources in a data center and implementing, to some extents, the distributed operating systems vision. At the same time, the prominent emergence of cloud computing has led to efforts towards inter-cloud architectures, aiming for better QoS, reliability and cost efficiency [37], [38]. This, in turn, fostered the expansion of this approach towards end users, introducing paradigms such as edge computing [3], [4] and fog computing [5]. Liquid computing extends these concepts towards a uniform infrastructural substratum for seamless distributed applications orchestration. Dynamism is a key distinguishing factor, enabling different pools of resources (e.g., cloud-based, cloudlets [39] and edge devices), likely under the control of different administrative domains, to transparently participate to one, or even multiple, computing continuums. Osmotic computing [40], given its broad scope, overlaps to some extents our proposal although, to the best of our knowledge, lacking a concrete characterization and targeting specifically the IoT domain.

The reminder of this section overviews the related work focusing on control plane and networking-related multi-cluster approaches in Kubernetes, including research proposals and relevant open-source projects. Indeed, we deem Kubernetes integration to be a key factor for smooth industry adoption. Proprietary solutions such as Google Anthos, Azure Stack, AWS Outposts and VMware Tanzu are out of scope, as bound to a specific environment, preventing the full realization of the computing continuum vision spanning across any *technological* and *administrative* domain.

A. Multi Cluster Kubernetes Control Plane

Kubernetes Cluster Federation (KubeFed) [41] represents the official community solution to the multi-cluster problem, and adopts a centralized approach to coordinate multiple clusters through an appropriate set of APIs. It implements a single point of control abstraction, as the users interact with the host cluster only, and the KubeFed control plane takes care of propagating the modifications. To overcome its limitations in terms of failure tolerance and number of supported clusters (as mostly focusing on large infrastructures), Larsson *et al.* [42] presented their vision towards a decentralized Kubernetes federation control plane. Their approach leverages a shared database of conflict-free replicated data types (CRDTs) to synchronize the global desired state, removing the single point of failure intrinsic in centralized solutions and foreseeing the support for thousands of federated edge clusters. Differently, Faticanti *et al.* [43] showcased the combination of KubeFed and their proprietary FogAtlas framework, the latter allowing to model distributed applications and automatically schedule the different components based on specified requirements. Still, all these solutions explicitly target infrastructures under the same administrative domain and focus on the control plane only, while requiring external solutions for inter-cluster communication.

The Karmada open-source project [44] adopted a different approach, leveraging a custom API Server to provide a single point of control abstraction, while mimicking the standard Kubernetes one. Then, vanilla high-level resources, such as Deployments, do not incur in the usual workflow, but get processed by the custom controllers and, depending on additional policy constraints defined through CRs, are eventually dispatched to the target clusters. Yet, this approach, which targets only the super cluster scenario, does not provide full Kubernetes compliance, preventing administrators the possibility to transparently deal with lower-level objects (i.e, pods) for accurate monitoring and troubleshooting operations. As an alternative solution, the *GitOps* paradigm [45] can also enable an elementary multi-cluster control plane, allowing the distribution of different tasks, described through declarative configuration files and automatically enforced by CI/CD mechanisms, across the entire infrastructure. Still, workload placement is completely static, lacking the possibility to dynamically migrate or scale applications across clusters to face unexpected failures or load spikes. Once more, this solution requires full control over the entire infrastructure, and it considers each cluster as an isolated silo, with no transparent communication support.

A different category of approaches leverages the VK abstraction to transparently masquerade the remote clusters, while introducing no API disruption and potentially supporting multi-ownership through proper isolation and permission limitations. Besides *liqo*, two main open-source projects followed this approach. First, *Admiralty* [46], a solution enabling different multi-cluster topologies, both centralized and distributed. It features a custom scheduling logic to select the best workload execution placement, complemented by resilient remote offloading through the custom *PodChaperon* abstraction and resource reflection. Second, *tensile-kube* [47], a project developed by Tencent Games to ensure high utilization in case of resources fragmented across multiple clusters. It supports remote pod offloading (although with no split-brain resiliency mechanisms), as well as resource reflection, along with custom scheduling and de-scheduling extensions to deal with resource fragmentation. Differently from *liqo*, tensile-kube resorts to external mechanisms for inter-cluster networking, as well as it poses no multi-ownership constraints as the entire infrastructure is assumed to be controlled by the same organization. Finally, the VK abstraction has also been leveraged in a different context by FLEDGE [48], a Kubernetes-compatible lightweight solution allowing individual low-resource edge devices to join an existing cloud-based cluster. Conversely, *liqo* targets multi-cluster scenarios, abstracting entire cluster slices as virtual nodes to enable seamless application scheduling, while adopting a decentralized approach to preserve the independence of each pool of resources.

B. Multi Cluster Kubernetes Network Interconnection

As for multi-cluster networking, we identified three main classes of solutions. First, CNI-provided (e.g., *Cilium ClusterMesh* [49]), hence featured by standard cluster connectivity components. However, this approach demands for coordination between all federating clusters to leverage the same technologies and prevent addressing conflicts, which is unsuitable in

case of dynamic, multi-ownership scenarios. Second, CNI-agnostic solutions, mainly including *Submariner* and *Skupper*. Submariner [50] enables cross-cluster layer-3 connectivity using encrypted VPN tunnels, while supporting service discovery and address conflict resolution mechanisms. Under the hood, it leverages a centralized, broker-based architecture to negotiate the interconnection configurations. Skupper [51], on the other hand, operates at layer 7 and possibly interconnects only a subset of remote Kubernetes namespaces, instead of the entire clusters they belong to. However, the above solutions address only the cross-cluster connectivity requirements, while leaving workload orchestration and observability across the entire resource continuum to either static approaches or other external tools. Third, service mesh-provided: a service mesh is a dedicated infrastructure layer for handling service-to-service communication, which is typically implemented as lightweight network proxies (i.e., sidecars) deployed alongside the actual applications [52]. Popular service mesh frameworks, including Istio [53] and Linkerd [54], feature also multi-cluster support through dedicated proxies, routing traffic from the mesh of one cluster to another. However, they do not implement a cross-cluster control plane, lacking a single point of entry to oversee the entire multi-cluster topology, and dynamically schedule the workloads in the best available location, regardless of the underlying infrastructure topology. In addition, service mesh solutions require complex configurations, and they come at a high cost in terms of application and latency overhead, due to the introduction of sidecars, which may be unsuitable especially in case of edge devices [55].

VIII. Conclusions

In recent years, the irrefutable success of the cloud and edge computing paradigms has brought to cluster proliferation in both small and large organizations, as well as the deployment of orchestrated edge devices running lightweight Kubernetes flavors. This trend inevitably leads to resource fragmentation, statically constraining applications execution to predetermined silos and introducing high operational complexity when fulfilling high availability requirements and policy compliance.

In this paper, we first advocated the opportunity for the introduction of liquid computing, a novel paradigm enabling a transparent continuum of computational resources and services on top of the underlying fragmented infrastructure. It foresees a decentralized, fluid and peer-to-peer architecture to account for the dynamism typical of edge and IoT devices, multi-ownership, to support the interconnection between different administrative domains, as well as an intent-driven approach, simplifying the characterization of each workload with high-level policies to constrain their execution to the most appropriate location. We believe liquid computing can simplify multi-cluster operations, through its intrinsic big cluster abstraction, and enable both resource sharing, to reduce allocation inefficiencies, and brokering scenarios, extending the highly successful IXP model to cloud and edge data center slices. Second, we presented and characterized *liqo*, an open-source project fostering the liquid computing vision through the creation of dynamic and seamless Kubernetes multi-cluster topologies. Extensive experimental evaluations have shown the effectiveness of *liqo*, both in terms of limited overhead with respect to vanilla Kubernetes and better performance compared to state of the art open-source solutions, while including at the same time more advanced features.

Acknowledgment

The authors would like to thank all the people who contributed to our journey towards the liquid computing vision, in particular Aldo Lacuku, Alessandro Olivero, Mattia Lavacca, Dante Malagrinò, all the students at Politecnico di Torino who collaborated on this project, and all the people who trusted *liqo* by running it on their production clusters.

This work was partly supported by European Union's Horizon Europe research and innovation programme under grant agreement No 101070473, project FLUIDOS (Flexible, scaLable, secUre, and decentralIseD Operating

References

[1] C. Pahl, "Containerization and the PaaS cloud," *IEEE Cloud Comput.*, vol. 2, no. 3, pp. 24–31, Jul. 2015.

[2] CNCF Staff, "CNCF annual survey 2021," Cloud Native Computing Foundation, Tech. Rep., 2021, (Retrieved: Mar. 2022). [Online]. Available: https://www.cncf.io/wp-content/uploads/2022/02/CNCF-AR_FINAL-edits-15.2.21.pdf

[3] P. Garcia Lopez *et al.*, "Edge-centric computing: Vision and challenges," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 5, pp. 37–42, Sep. 2015.

[4] W. Shi *et al.*, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.

[5] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proc. 1st Edition ACM MCC Workshop on Mobile Cloud Computing*, Aug. 2012, p. 13–16.

[6] D. Milojicic, "The edge-to-cloud continuum," *IEEE Computer*, vol. 53, no. 11, pp. 16–25, nov 2020.

[7] L. Baresi, D. F. Mendonça, M. Garriga, S. Guinea, and G. Quattrocchi, "A unified model for the mobile-edge-cloud continuum," *ACM Trans. Internet Technol.*, vol. 19, no. 2, apr 2019.

[8] M. A. Tamiru, G. Pierre, J. Tordsson, and E. Elmroth, "Instability in geo-distributed kubernetes federation: Causes and mitigation," in *28th Int. Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Nov. 2020, pp. 1–8.

[9] L. Larsson, W. Tärneberg, C. Klein, E. Elmroth, and M. Kihl, "Impact of etcd deployment on kubernetes, istio, and application performance," *Softw Pract Exp*, vol. 50, no. 10, pp. 1986–2007, Aug. 2020.

[10] L. Osmani, T. Kauppinen, M. Komu, and S. Tarkoma, "Multi-cloud connectivity for kubernetes in 5g networks," *EEE Commun. Mag.*, vol. 59, no. 10, pp. 42–47, Oct. 2021.

[11] L. Leong, "Comparing cloud workload placement strategies," Gartner Research, Tech. Rep., 2020, (Retrieved: Mar. 2022). [Online]. Available: https://www.gartner.com/en/documents/3990249/comparing-cloud-workload-placement-strategies

[12] D2IQ, "Multi-cluster management: Reduce overhead and redundant efforts," D2IQ, Tech. Rep., 2021, (Retrieved: Mar. 2022). [Online]. Available: https://d2iq.com/resources/cheat-sheet/multi-cluster-management-reduce-overhead-and-redundant-efforts

[13] P. Mell and T. Grance, "The NIST definition of cloud computing," NIST, Tech. Rep., Sep. 2011, (Retrieved: Mar. 2022). [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf

[14] A. Yousafzai *et al.*, "Cloud resource allocation schemes: Review, taxonomy, and opportunities," *Knowl. Inf. Syst.*, vol. 50, no. 2, p. 347–381, Feb. 2017.

[15] R. Buyya *et al.*, "A manifesto for future generation cloud computing: Research directions for the next decade," *ACM Comput. Surv.*, vol. 51, no. 5, pp. 1–38, Nov. 2018.

[16] P.-J. Maenhaut, B. Volckaert, V. Ongenae, and F. D. Turck, "Resource management in a containerized cloud: Status and challenges," *J Netw Syst Manage*, vol. 28, no. 2, pp. 197–246, Apr. 20.

[17] G. Gruman, "Welcome to the next tech revolution: Liquid computing," https://www.infoworld.com/article/2608440/article.html, Jul. 2014, (Retrieved: Mar. 2022).

[18] R. Bias, "Architectures for open and scalable clouds," https://www.slideshare.net/randybias/architectures-for-open-and-scalable-clouds, Feb. 2012, (Retrieved: Mar. 2022).

[19] CNCF Staff, "CNCF survey 2020," Cloud Native Computing Foundation, Tech. Rep., 2020, (Retrieved: Mar. 2022). [Online]. Available: https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf

[20] "K3s: Lightweight kubernetes," https://k3s.io, (Retrieved: Mar. 2022).

[21] K. Goldenring, "Announcing akri, an open source project for building a connected edge with kubernetes," https://cloudblogs.microsoft.com/opensource/2020/10/20/announcing-akri-open-source-project-building-connected-edge-kubernetes/, Oct. 2020, (Retrieved: Mar. 2022).

[22] Z. Zheng *et al.*, "An overview on smart contracts: Challenges, advances and platforms," *Future Gener. Comput. Syst.*, vol. 105, pp. 475–491, Apr. 2020.

[23] J. Fritsch and C. Walker, "The problem with data," in *Proc. 2014 IEEE/ACM 7th Int. Conf. on Utility and Cloud Computing (UCC)*, Dec. 2014, pp. 708–713.

[24] S. Cheshire and M. Krochmal, "Dns-based service discovery," RFC Editor, RFC 6763, Feb. 2013.

[25] "Virtual kubelet," https://github.com/virtual-kubelet/virtual-kubelet, (Retrieved: Mar. 2022).

[26] J. Donenfeld, "Wireguard: Next generation kernel network tunnel," in *Proc. Network and Distributed System Security (NDSS) Symp.*, Feb. 2017, pp. 1–12.

[27] "Controller runtime," https://github.com/kubernetes-sigs/controller-runtime, (Retrieved: Mar. 2022).

[28] M. Hausenblas and S. Schimanski, *Programming Kubernetes*. O'Reilly Media, Inc, Jul. 2019.

[29] "Kubemark user guide," https://github.com/kubernetes/community/blob/master/contributors/devel/sig-scalability/kubemark-guide.md, (Retrieved: Mar. 2022).

[30] "Kubernetes: Considerations for large clusters," https://kubernetes.io/docs/setup/best-practices/cluster-large, (Retrieved: Mar. 2022).

[31] "K3s: Installation requirements," https://rancher.com/docs/k3s/latest/en/installation/installation-requirements, (Retrieved: Mar. 2022).

[32] S. Böhm and G. Wirtz, "Profiling lightweight container platforms: Microk8s and k3s in comparison to kubernetes." in *Proc. ZEUS Workshop*, Feb. 2021, pp. 65–73.

[33] M. Iorio, A. Palesandro, and F. Risso, "CrownLabs — a collaborative environment to deliver remote computing laboratories," *IEEE Access*, vol. 8, pp. 126 428–126 442, Jul. 2020.

[34] A. S. Tanenbaum and R. Van Renesse, "Distributed operating systems," *ACM Comput. Surv.*, vol. 17, no. 4, p. 419–470, Dec. 1985. [Online]. Available: https://doi.org/10.1145/6041.6074

[35] A. S. Tanenbaum, "Distributed operating systems anno 1992. what have we learned so far?" *Distrib. Syst. Eng.*, vol. 1, no. 1, pp. 3–10, Sep. 1993.

[36] J. Gosling and H. McGilton, "The Java language environment: A white paper," Sun Microsystems, Inc, Tech. Rep., Oct. 1995, (Retrieved: Mar. 2022). [Online]. Available: https://www.stroustrup.com/1995_Java_whitepaper.pdf

[37] N. Grozev and R. Buyya, "Inter-cloud architectures and application brokering: taxonomy and survey," *Softw. - Pract. Exp.*, vol. 44, no. 3, pp. 369–390, Mar. 2014.

[38] A. N. Toosi, R. N. Calheiros, and R. Buyya, "Interconnected cloud computing environments: Challenges, taxonomy, and survey," *ACM Comput. Surv.*, vol. 47, no. 1, May 2014.

[39] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE Pervasive Comput.*, vol. 8, no. 4, pp. 14–23, Oct. 2009.

[40] M. Villari *et al.*, "Osmosis: The osmotic computing platform for microelements in the cloud, edge, and internet of things," *IEEE Computer*, vol. 52, no. 8, pp. 14–26, Aug. 2019.

[41] "Kubefed: Kubernetes cluster federation," https://github.com/kubernetes-sigs/kubefed, (Retrieved: Mar. 2022).

[42] L. Larsson, H. Gustafsson, C. Klein, and E. Elmroth, "Decentralized kubernetes federation control plane," in *Proc. 2020 IEEE/ACM 13th Int. Conf. on Utility and Cloud Computing (UCC)*, Dec. 2020, pp. 354–359.

[43] F. Faticanti, D. Santoro, S. Cretti, and D. Siracusa, "An application of kubernetes cluster federation in fog computing," in *Proc. 24th IEEE Conf. on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, Mar. 2021, pp. 89–91.

[44] "Karmada," https://github.com/karmada-io/karmada, (Retrieved: Mar. 2022).

[45] T. A. Limoncelli, "Gitops: A path to more self-service it: Iac + pr=gitops," *ACM Queue*, vol. 16, no. 3, p. 13–26, Jun. 2018.

[46] "Admiralty," https://admiralty.io, (Retrieved: Mar. 2022).

[47] W. Cai and Y. Yin, "Extending kubernetes to an unlimited one through tensile-kube," https://www.cncf.io/blog/2020/08/11/extending-kubernetes-to-an-unlimited-one-through-tensile-kube/, Aug. 2020, (Retrieved: Mar. 2022).

[48] T. Goethals, F. DeTurck, and B. Volckaert, "Extending kubernetes clusters to low-resource edge devices using virtual kubelets," *IEEE Trans. on Cloud Comput.*, pp. 1–15, Oct. 2020 - To appear.

[49] "Deep dive into cilium multi-cluster," https://cilium.io/blog/2019/03/12/clustermesh, Mar. 2019, (Retrieved: Mar. 2022).

[50] "Submariner," https://submariner.io, (Retrieved: Mar. 2022).

[51] "Skupper," https://skupper.io, (Retrieved: Mar. 2022).

[52] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, "Service mesh: Challenges, state of the art, and future research opportunities," in *Proc. 2019 IEEE Int. Conf. Service-Oriented System Engineering (SOSE)*, Apr. 2019, pp. 122–1225.

[53] "Istio," https://istio.io, (Retrieved: Mar. 2022).

[54] "Linkerd," https://linkerd.io, (Retrieved: Mar. 2022).

[55] D. Espinel Sarmiento, A. Lebre, L. Nussbaum, and A. Chari, "Decentralized SDN control plane for a distributed cloud-edge infrastructure: A survey," *IEEE Commun. Surv. Tutor.*, vol. 23, no. 1, pp. 256–281, Jan. 2021.