



A geographical-aware state deployment service for Fog Computing

Diogo Lima^{a,b,*}, Hugo Miranda^b

^a Escola Superior de Hotelaria e Turismo do Estoril, Portugal

^b LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal

ARTICLE INFO

Keywords:

Mobile distributed applications
Fog Computing
Geographical state deployment
Consistency
Lightweight groups

ABSTRACT

Today's distributed mobile applications assume a permanent network connectivity to mediate the interactions between a large number of users, coordinated by resourceful servers on cloud datacenters. Cloud-based distributed applications penalize performance with the unavoidable latency and jitter resulting from the geographical distance between clients and application servers. Fog Computing architectures mitigate this impact by deploying state fragments and providing computing power on surrogate servers, located at the network edge. Performance improves with the ability to deploy each fragment at the most convenient surrogate, and with efficient consistency procedures even when fragments at different locations are used. This paper presents a self-configuring geographical-aware state deployment service that combines a state deployment algorithm with a scalable distribution support framework.

1. Introduction

Applications for mobile devices can present different characteristics that make them fall into one of two categories. They can either be independent of Internet connectivity and make exclusively use of the computing power and data available at the devices, or condition their execution to the availability of a network connection. The former is characterized by the avoidance of network bottlenecks, but the service is restricted by the limited computing power and memory of the mobile devices. The latter leverages on the almost permanent Internet connectivity of smartphones and on the flat rate charging model to support the current market trend of ignore or, at least neglect, the impact of using the network. This trend has fostered a change in mobile users' behaviour, who increasingly produce and demand larger amounts of data. For instance, the International Data Corporation (IDC) 2021 Global DataSphere forecast [1] reported that 64.2ZB of data was created or replicated in 2020 and the expectation for the next five years is that the amount of digital data created will be greater than twice the amount of data created since the advent of digital storage. Networked applications are thus vulnerable to the increasing network traffic and to the Internet's best effort quality of service principle. To avoid massive upstream data uploads, consuming core network resources and introducing latencies, solutions have proposed to bring computing and storage power to the edge of the network, deciding the best placement for edge servers [2].

Distributed mobile applications are one case of this second category. Examples include collaborative applications for smart cities and traffic

management [3], massive online augmented reality games (e.g. Ingress, Pokemon Go), social networks (e.g. Twitter, Foursquare) and tourism applications to enhance travellers' experience (e.g. Yelp, Culture Trip). The benefits of using these applications may vary, but often include the ability to collaborate in sensing the surrounding physical environment, discover and disseminate content, and support decisions based on past experiences of others. In all these cases, distributed mobile applications manage the data that users consume and produce. This data is hereafter referred as the *application state* and is assumed to be composed by shared objects that users can create, read, update and delete.

The current trend in the implementation of distributed mobile applications is to simplify state management by centralizing it in large scale datacenters located in the Cloud. However, cloud datacenters aim at a continent-wide coverage, which creates a non-negligible physical distance between the application state and the users. As this distance may extend for possibly thousands of kilometres, users are exposed to high latency and uncertain response times (jitter), due to the competition for the available bandwidth on backbone networks [4]. This paper proposes to attenuate the negative impact of this geographical barrier between the application state and the users, which results in a degradation of distributed mobile applications performance.

Fog Computing proposes to overcome this problem by approximating the Cloud from the users with the deployment of *surrogate servers* at the edge of the network, in locations such as bus terminals, shopping malls, public offices [5] or even underwater [6]. Surrogates provide computing power and memory to manage application state, going one

* Corresponding author at: LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal.

E-mail addresses: dmslima@ciencias.ulisboa.pt (D. Lima), hamiranda@ciencias.ulisboa.pt (H. Miranda).

step beyond the caching (read-only) service provided by Content Delivery Networks (CDN). The judicious distribution by surrogate servers of fragments of the application state, originally concentrated in cloud datacenters, gives to Fog Computing the potential to mitigate latency, as well as to reduce load in the backbone networks [7]. However, to achieve it, two problems that could not be found in the centralized Cloud approach must be addressed.

The first is hereafter referred as the *geo-aware state deployment* (GeoSD) problem. Fog Computing can only mitigate latency and backbone traffic if it is able to deploy each of the application state fragmented components on the surrogate at the most favourable location. The deployment algorithm must consider that, even within a simple application, state components do not behave equally. Some components should follow the user (such as one's daily calendar), a concept hereafter coined as *nomadic state*. Others are directly related to some physical location and only users in proximity are likely to be interested in accessing and manipulating it. This is the case of data related with specific landmarks in a city, and is hereafter referred as *location-dependent state*. The problem is amplified by noting that applications manage these components in conjunction. For example, a rating of a restaurant in Foursquare (location-dependent state) implies an update to the rater's profile (nomadic state). The deployment algorithm should avoid that components accessed together end up deployed in distant surrogates, resulting on higher latency and jitter. Moreover, as data utilization may evolve during the application's lifetime, deployment must be dynamic and reassessed periodically. This, in turn, requires mechanisms coordinating data migration among surrogates and the ability to retrieve the current location of each application state component. In summary, the location management problem is composed of three challenges: (i) decide the most suitable location for state components, (ii) coordinate their migrations and (iii) facilitate their location by other surrogates.

The second problem, hereafter referred as the *distributed environment* problem, is concerned with state consistency, in particular when operations require the participation of multiple surrogates. As the application state moves from a centralized management approach of the Cloud to a widespread deployment in Fog Computing, it will become fragmented among different surrogates. These surrogate will manipulate application state components that might be stored locally or in different surrogates, in concurrent fashion with operations issued by other surrogates, so it is important to guarantee that all these participants observe the same state over an application component. In turn, these surrogates are susceptible to crash or suffer network partitions. Hence, Fog Computing needs to provide applications with a scalable framework to maintain global state consistency, that is able to cope with application state fragmentation and the possibility of surrogate crashes or network disturbances, without compromising performance.

State of the art of distributed fog computing applications trade off consistency by performance by adopting optimistic eventual consistency models that tolerate some state inconsistencies. This paper tries to close the gap between consistency and performance. Expectations are that the advent of 5G will lead to a proliferation of applications with stringent consistency and timeliness requirements, consuming massive amounts of data. Examples can be found on applications so simple as assuring that traffic lights of an intersection are consistently observed by all (possible autonomous) vehicles approaching it, even in the presence of vehicles with the right-of-way.

This paper proposes a platform that provides a data management service for Fog Computing, that includes:

- deploying application state closer to where it is more frequently used,
- providing strong consistency.

To achieve these goals, the service relies on algorithms that monitor application state utilization and decide migrations in run-time as well as virtual synchrony protocols to ensure consistency.

2. Problem statement and system model

This paper considers scenarios such as the one exemplified by a large-scale forest surveillance system, depicted in Fig. 1, where heterogeneous devices (ground sensors, thermal cameras, UAVs, etc.) rely on a permanent support infrastructure for early detection of wild fires, hikers search and rescue or illegal logging. The support infrastructure is composed of surrogate servers (hereafter named the *smart Ground Stations* (GS) for simplicity), deployed in advance over the surveyed site and connected to the Internet. Server placement in the surveyed area may be determined by algorithms such as PACK [2], optimizing the distances between servers and their associated access points. The various devices can communicate with the GS using any available network technology and protocols. Besides collecting the large amount of data the devices produce and forwarding it for data analytics applications, GS are also required to make split second decisions, for example to divert UAVs and humans from their predefined route in order to investigate a smoke alarm. Moreover, as some operations must be coordinated with nearby GS (i.e. if the rerouting of UAVs has the potential to create mid-air collisions) and in order to ensure application availability in face of damages or power outages, GS are connected to each other and state components are replicated in nearby GS.

This paper assumes that application state can be partitioned in 3 categories. *Global* state is the information that is relevant to all participants in the system. In the forest surveillance example, global state refers to data such as the unresolved alarms and the location of GSs. *Location-dependent* state refers to data which is relevant to the participants located in a specific area and unlikely to be accessed by more distant participants. Good examples of location-dependent state are the set of measurements registered by sensors (i.e. humidity levels, temperature, wind speed), as well as the current number and location of UAVs and humans in that area. Finally, *nomadic* state refers to data only relevant to specific mobile devices, such as the current battery level, flight autonomy and operational status.

Inspired by the forest surveillance system, this paper defines a large scale fault tolerant distributed system composed by surrogates that manage a fraction of the application state. A geo-aware state deployment strategy is implemented, allowing each state item to be stored at the surrogate most likely to reduce long-distance accesses and therefore latency. State deployment is managed by a service, that conceptually acts as an oracle, with full knowledge of the system dataset and current deployment. State item utilization awareness is achieved by collecting transaction logs (for example the resolution of an alarm), delivered in the background by the surrogates. Periodically, the oracle re-evaluates the current deployment, decides the most suitable location of each state item and coordinates its transfer between surrogates. Since the deployment evaluation is a resource intensive operation, it is triggered whenever some predefined number of transaction logs is received.

Application transactions may affect more than one state item, possibly hosted by distinct surrogates. Depending on the number of sites involved, transactions can be either *local* or *distributed*. Local transactions are those that access data stored at a single surrogate (e.g. calling a nearby UAV back to a GS for battery charge), while distributed transactions involve multiple surrogates. A good example of a distributed transaction is the request of a GS for more devices to help estimate the extension and progression speed of an existing fire. A local transaction is handled exclusively by the surrogate. The complexity and latency of coordinating distributed transactions motivate the tendency to keep commonly accessed state components at the same surrogate.

Nevertheless, even the more judicious geo-aware state deployment policy will not be able to achieve the ideal scenario where each transaction can be processed by a single surrogate. Whether because nomadic state has not been migrated yet, because state items are being evenly accessed and disputed by multiple surrogates, or by the simple fact that a transaction may involve a significant amount of state items located in different deployment sites, many reasons may contribute to the fact

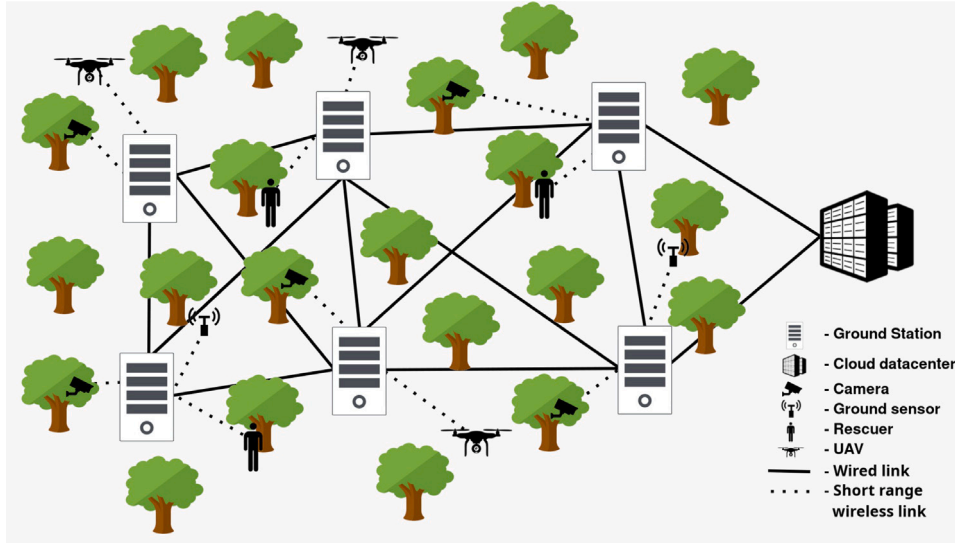


Fig. 1. Example of App.

that distributed transactions will be unavoidable. Moreover, distributed transactions are also a consequence of the need to replicate data to tolerate hardware and network faults. Therefore, each application state component is assumed to be replicated by a few surrogates. The greater the number of replicas, the more fault tolerance is provided at the cost of scalability. Hence, a partial replication scheme that deploys each component on a subset of surrogates is followed.

Replication hardens the consistency challenge as more participants are involved in each transaction. Current solutions for distributed Fog Computing applications prioritize efficiency and availability over consistency, relaxing it to optimistic eventual consistency models provided by NoSQL systems. To provide strong consistency, as required by our forest surveillance system, application developers need to implement additional layers at the middleware level or rely on datastores that support transactions such as Google Spanner [8] and COPS [9], located in the Cloud and susceptible to latency overheads.

To address consistency, our system model revisits the virtual synchrony abstraction [10]. For such, the surrogate servers replicating each component are conceptually organized as *groups*, i.e., for some state item named *A*, all surrogates that store a replica of *A* form the *group* of *A*. Together, members of the group of *A* are responsible for validating all transactions that manipulate *A*, ensuring state consistency. Transactions that manipulate multiple state items require the creation of larger groups, that combine the members of each group of items involved in the transaction. Therefore, we propose a consistent multi group Fog Computing system model, where the challenge lies in coping with the large number of groups and members.

3. Related work

Tackling the geo-aware state deployment (GeoSD) and the distributed environment problem combines goals of two distinct active research fields. This section makes a brief discussion on the most relevant research results on data partition and deployment, and on coordination and consistency.

3.1. Data partition and deployment

The challenge posed by the GeoSD problem presents some similarities with caching, in particular with Content Delivery Networks (CDNs) whose goal is to geographically distribute replicas of the most popular content as a way to balance servers' load and reduce latency. However, CDNs and caching in general only consider read-only replicas of data

and trivially resolve the data consistency problem by performing every update in the primary replica and invalidating cached copies [11,12]. In contrast, the GeoSD problem aims at providing state consistency with concurrent updates, identifying the ideal location for each single copy of the data and knowing that data can be edited and new content created by clients. GeoSD must minimize latency of data access, regardless of the type of operation (read or write) performed over the data. Additionally, while in CDNs each server can take caching decisions individually, in GeoSD the system needs to be considered as a whole as write operations are likely to involve applications state items stored in different servers.

3.1.1. Data partitioning

The GeoSD problem thus relies on partitioning the application state in different components and then decide the best location for each of those components. Data partitioning was originally proposed to evenly balance the load across the multiple servers composing a database system as a way to improve scalability and performance. Horizontal and vertical partitioning are some of the earliest strategies to emerge in data partitioning [13], to make the amount of data more manageable at each server by aggregating sets of rows (horizontal) or columns (vertical) of a table. Graph partitioning, on the other hand, considers the implicit association created between state items based on transactions. Database items are partitioned by storage nodes so that most transactions only need to access one node [14], reducing locking and consistency overhead. To achieve this goal, data items are mapped onto vertexes while edges are used to represent associations between items in one or more transactions. The mapping of state items into nodes is then performed by applying a graph partitioning algorithm such as *k*-way min/cut [15]. Unfortunately, data partitioning in GeoSD focuses on identifying data's geographical relevance so that it can be deployed where it is more frequently used, rather than load balancing.

Beyond database management systems, data partitioning has also been exploited to improve scalability and throughput of distributed applications. ZooFence [16] uses the Apache ZooKeeper [17] coordination service to build dependable and consistent partitioned services. Each partition elects a leader and has an atomic queue. Once the service is partitioned, clients can either (i) issue a command directly to a single ZooKeeper instance if that command can be treated by a single instance or (ii) insert a command on the appropriate partition queue. In the latter case, an *executor* component is responsible to forward the commands to the associated ZooKeeper instances, merge their results and reply to the client. Compared to the GeoSD problem,

ZooFence suffers from the limitation of considering partitions to be immutable: once the partitions are defined, state components cannot migrate among partitions. Instead, a GeoSD approach must be able to migrate application state components as the usage patterns may change.

State mobility can be found in AdaptCache [18], a cooperative and integrated cache framework for web enterprise systems. AdaptCache uses an “oracle” which dynamically evaluates and manages state items placement. Unfortunately, requests are once again distributed across the servers to achieve load balancing and simplify consistency management, rather than identifying data’s geographical relevance. The same is observed in other systems such as [19–21], where the main objective lies on maintaining load balancing among partitions. In summary, current data partitioning strategies focus on load balancing and are oblivious to the geographical relevance of the application state. GeoSD, on the other hand, must partition the application state into components and then identify, for each, where they are more frequently used. Even if load imbalances may emerge in the system.

3.1.2. Overlay network deployment in P2P

In peer-to-peer (P2P) systems an overlay network is deployed on top of an existing physical network. The nodes of the physical network are organized to form a logical topology composed of overlay links, and the peers communicate with each other at the application level. Structured overlay networks rely on decentralized construction protocols [22,23] to organize the peers into a specific topology. However, most popular overlay networks do not take into consideration the infrastructure on which the overlay is deployed. Based on this problem, Fluidify [24] and others proposed an approach for overlay creation that considers data placement in the system. Fluidify follows a dual strategy that exploits both the logical links of the overlay and the physical topology of the underlying infrastructure to progressively align one with the other in order to achieve network locality.

This approach shares with GeoSD the goal of considering network locality as a tool to reduce latency. However, in Fluidify this is done at the node link level by dynamically re-shaping the overlay network topology so that the peers have the most suited set of neighbours, aligned with the underlying physical infrastructure. Here, we are focused on application data and on determining the most suitable server to store it so that it better fits the usage pattern observed.

3.1.3. Geo-replicated cloud storage

In contrast with the previous geo-oblivious approaches, geo-replicated storage solutions incorporate the notion of physical distance into data partitioning. These usually focus on tolerating failures at the datacenter level, while providing some approximation of the application state replicas stored in the cloud to the users spread worldwide.

Volley [4] tries to minimize the user-perceived latency to access data stored in the cloud while respecting the associated business constraints. Volley relies on clients’ access logs to determine data location. Its data placement algorithm starts by translating client user IP addresses into geographic coordinates. Then, each data item is also mapped into geographic coordinates based on the weighted average of the geographic coordinates for the client IP addresses that access it. Volley then iteratively measures the communication frequency and latency between data items and applies an attracting force to pull often correlated objects together. Finally, data items are deployed to the datacenter closest to its calculated geographic coordinates. Volley’s algorithm gives higher priority to the *location-to-object* association than to the *object-to-object* association. Therefore, it will likely under-perform in face of highly correlated datasets where the most important aspect is to maintain the *object-to-object* association intact regardless of the multiple locations clients accessing them.

Geographic replication has also been addressed to improve other metrics, like reliability. Tuba [25] is a geo-replicated key-value store

that automatically reconfigures its set of replicas to meet with the SLAs required by its clients. CockroachDB¹ is a distributed transactional SQL database where replica location is decided automatically, based on user configured criteria such as the types of failures to tolerate. Unfortunately, CockroachDB cannot encompass any concerns related with client access latency.

Regardless of the data partitioning strategy used, multi partition transactions are by themselves a complex challenge that negatively impacts system performance. DS-SMR [26] implements a dynamic and scalable replicated state machine that transfers all required data items to a single partition prior to initiating every transaction. However, its performance can be penalized by the non-negligible latency resulting from state item migrations during transaction execution.

Workload analysis and location manipulation can be found in the SDUR [27] geo-replicated storage system. Authors consider that local transactions can be validated and committed faster than distributed transactions, and that in the presence of mixed workloads, a local transaction delivered after a global transaction will experience a longer delay. SDUR reorders transactions to give priority to those using a single partition and deferring global transactions. In contrast, the solution presented in this paper aims at reducing the overall number of global transactions.

3.1.4. Data deployment in Cloudlets, OEC, IoT and Fog Computing

More recently, with the advent of computing models that approximate servers to end users, rather than being located in the distant Cloud, specific solutions have been addressed to decide the placement of servers at the network edge. Cloudlets refer to surrogate servers deployed at the network edge where users can deploy virtual machine images to execute their tasks. Communication between cloudlet servers is not expected and users are assumed to have a one-to-one relation with the cloudlet server where their VM image has been deployed. Hence, an initial cloudlet placement is needed to provide users with available computing power in the most required locations [28,29]. Unfortunately, since Cloudlets are oblivious to any application state, their deployment objectives are not comparable with those of the GeoSD problem.

On the other hand, Opportunistic Edge Computing (OEC) differs from the other categories of edge computing as it considers that the system formed at the network edge is composed of mobile devices, without a permanent support of a fixed infrastructure of surrogate servers. Hence, the problem of deploying application state in this model is conditioned by the characteristics of the opportunistic networks that are created among mobile devices as they encounter each other. In such scenario, rather than deciding the deployment of application state to reduce latency and improve performance, such as the GeoSD problem, the main objective is to provide availability and avoid application state partitions to users. State deployment is thus dependent on the mobile devices’ mobility patterns and the probability they have in contacting each other along time [30].

In IoT, the infrastructure of surrogate servers is deployed to support and process the data generated by sensors monitoring the physical world. The main difference between IoT and Fog Computing is that, in IoT, the devices that generate data can be seen exclusively as producers (sensors do not need the results of data processing). In the Fog Computing, mobile devices are considered producers, consumers and manipulators: they generate state, but also manipulate and interact with it. Hence, the deployment of surrogate servers for IoT solutions can be translated by how many and where to deploy surrogate servers so that latency in sensor data transmission is minimized [31]. Deployment decisions are influenced by constraints such the number of sensors attached to each surrogate server, but are oblivious to any data processing and analysis occurring in surrogate servers. This is contradicting to

¹ <https://www.cockroachlabs.com/>

the GeoSD problem, where the latency to minimize focuses precisely on the application state: where it is most often manipulated and with which state components one is most likely used in conjunction.

Therefore, the work closely related to GeoSD is the PACK [2] placement algorithm. PACK is a server placement algorithm for Fog Computing that, based on historic workload information, tries to minimize the latencies between users and surrogate servers. PACK differs from the objectives pursued in the GeoSD problem on three points. The first is that PACK considers a fixed number of surrogate servers a priori which remain constant throughout. Conversely, the GeoSD problem is agnostic to the number of existing surrogates. Depending on the number of surrogates available, the GeoSD will be able to output more refined or less refined state deployments, but it can run with a varying number of surrogates along time. Secondly, as many other examples referred so far, PACK's main deployment objective is to achieve load balancing in the components deployed to each surrogate. This is a fundamental difference to GeoSD, where unbalanced scenarios can arise in Fog Computing and the deployment algorithm must be able to exploit it if it ultimately benefits system performance, instead of mitigating it. Finally, PACK does not consider application state migrations after the initial deployment, making it impossible to react to possible usage patterns changes and user mobility, which the GeoSD considers.

3.2. Coordination and consistency

In distributed systems, data consistency can be delegated by surrogates on centralized Cloud-based systems [32], or managed locally at the network edge [33]. As we consider that the benefits of the Fog Computing model are better exploited when application state is stored and managed closer to the end users, the study for the related work focuses on solutions that can be applied at the network edge.

3.2.1. Group view synchronous communication

State Machine Replication (SMR) [34] is an abstraction to implement a fault-tolerant service where data is replicated in multiple servers. SMR is characterized by the simple rationale of applying the same sequence of deterministic operations to a set of state machines with the same initial state, leading to the same final state on every state machine. Challenges are on ensuring that the same operations are delivered in the same order to every replica in a faulty environment, a concept coined as Atomic Broadcast. Since SMR aims to improve exclusively the systems' tolerance to faults and knowing that every replica executes every operation, it can be assumed that the system performance is dictated by the slowest of its members. In addition, it has also been shown that atomic broadcast requires a consensus algorithm, such as Paxos [35] or Raft [36]. Unfortunately, consensus algorithms are well-known for their scalability problems.

Addressing the scalability limitations of SMR necessarily implies a compromise elsewhere. DS-SMR [26] distributes state components by system partitions, composed by one or more participants. The state is kept consistent by delegating the execution of each operation to a single partition. However, DS-SMR requires state items to be transferred between partitions, in order to ensure that the most up-to-date version of each state item is used on every operation, favouring performance penalties from the frequent migration of state components.

DPaxos [37] is a variant of Paxos intended to be used as the SMR component for Fog Computing systems. Scalability is addressed by proposing smaller and dynamic leader election quorums. It considers a geographically distributed setting divided into zones, composed of disjoint sets of nodes, and proposes *Zone-centric quorums* instead of Paxos' majority-based quorums to avoid wide-area communications and reduce latency. The objective is to achieve leader election quorums as small as a single zone and expand as conflicts are detected. Therefore, DS-SMR and DPaxos are optimized for workloads where operations do not frequently access objects in multiple partitions. Unfortunately,

Table 1

Group view synchronous communication specification.

Module	
Name:	View Synchronous Communication, instance vs
Properties:	
VS1 (View Inclusion)	If some process delivers a message m from process p in view V , then m was broadcast by p in view V .
VS2 (Validity)	If a correct process p broadcasts a message m , then p eventually delivers m .
VS3 (No duplication)	No message is delivered more than once.
VS4 (No creation)	If a process delivers a message m with sender s , then m was previously broadcast by process s .
VS5 (Agreement)	If a message m is delivered by some correct process, then m is eventually delivered by every correct process.

applications with other usage patterns will be faced with the problem of constantly changing data placements between partitions (DS-SMR) or expanding the leader election quorums to the full system size (DPaxos). This reduces the locality gains, converging to systems comparable to the original SMR.

Virtual synchrony [10] organizes processes into groups, simplifying SMR implementation by restricting the fault model. The concept of *view* is key for the implementation of virtual synchrony. A view is defined by an ordered list of the correct processes of the group [38] and a monotonically increasing ID number. The ordered list of correct processes implicitly defines a view leader. The view ID allows each message to be uniquely associated to a view. Within each view, messages must be delivered to all correct group view members, or to none at all. Views are managed by a membership service, in charge of detecting faulty processes, accept join requests and advertise (install) new views once the group membership changes.

Table 1 depicts the view-synchronous communication abstraction properties as presented in [39]. Properties VS2 to VS5 provide delivery guarantee to messages broadcast by a correct process. Property VS1 defines a boundary for the reliable broadcast message delivery, closing the list of recipients to the correct processes that were members of the view where the message was sent. In practice, view synchrony provides a simple abstraction to facilitate inter-process communication and state synchronization given that if a process installs two consecutive views V_1 and V_2 it was necessarily correct on V_1 and therefore received all the messages reliably broadcast in V_1 . To recover from a failure, the system simply needs to keep track of the latest view each process installed and implement some synchronization protocol once a view is installed. Atomic broadcast [40] simplifies the implementation of state machine replication trading off performance by scalability. Gossip multicast [41] and optimistic virtual synchrony [42] are examples of proposals that loosen some of the strict requirements for the implementation of state machine replication increasing scalability but also the burden on the application programmer to provide consistency.

Different approaches to implement virtual synchrony have been proposed (Ensemble, Horus, Cactus, to name a few). For example, Spread [43] relies on daemons to create virtual synchronous groups and the group communication services are all concentrated in the daemon. If multiple processes are attached to the same daemon and the latter crashes, all processes attached to it are affected. Appia [44], on the other hand, follows a decentralized approach. Therefore, a crash only disconnects that process from the virtual synchrony service, facilitating recovery.

3.2.2. Consistent data types

Consistency can also be achieved at the datatype level. A Conflict-free Replicated Data Type (CRDT) [45] ensures that all replicas that received the same set of updates (even if in a different order) converge to the same final state. Changes are announced either by propagating

the updated state and merging it into remote copies (state-based), or by propagating the operations and applying them (operation-based). A convergence service is responsible for guaranteeing that replicas end up with the same state after all of the updates and merges have been received. Although CRDTs have been used in real-world scenarios of stateful computations at the network edge [46], they suffer from major adaptability limitations that turn them unsuitable for general purpose applications, for example, by restricting the number of usable data types [47]. In addition, CRDTs cannot cope with the consistency requirements that emerge from updating multiple state components.

3.2.3. Coordination in internet-scale systems

Apache ZooKeeper [48] is currently one of the most popular services for coordinating processes in distributed applications. ZooKeeper exposes an API to a coordination service that provides to the clients the abstraction of a set of data objects, called *Znodes*, organized in a hierarchical tree structure that stores coordination information. To provide high availability, ZooKeeper replicates data on every server that composes the service. While read requests are executed locally by the server to which the client is connected, write requests are forwarded to the leader instance. The latter executes the request and propagates the changes to the other replicas through the Zab [49] atomic broadcast protocol.

By creating *Znodes* to store application state specific to a group and ephemeral *znodes* to implement group membership, ZooKeeper is a good candidate for implementing our consistent multi group Fog Computing system model. Implementation would create and manage groups using multiple ZooKeeper tree instances so that surrogates could create new ephemeral nodes in the tree coordinating the groups they want to join. However, such solution suffers of scalability problems as one surrogate can be elected leader of multiple groups. Moreover, since each group would be coordinated by a different ZooKeeper tree instance, it is possible that one process is considered faulty in some groups and correct in others, what may result in challenging consistency problems. In contrast, the approach presented in this paper guarantees that, as long as none of the processes crash, reliable broadcast is sufficient to perform write operations and, as multiple groups are projected onto a coarser virtual synchrony group, if a process is detected as faulty, this information remains consistent among all groups.

PathStore [50] considers a session consistency model where the mobile client establishes a session with the closest replica and consistency is maintained in all operations performed within that session. To cope with mobility, whenever the client changes location and reaches another replica, the latter contacts the initial replica that created the session to perform the synchronization of manipulated data. New operations are halted until this reconciliation procedure is finished. This system follows a reactive approach where all application state manipulated within a session is relayed among replicas as clients are observed to change locations. This is in contrast with our solution, which relies in proactively creating groups of processes that replicate the same application state, avoiding costly reconciliation procedures during client execution of an operation. The notion of groups of replicas is also exploited in FogStore [51], a key-value data store designed for low-latency accesses in the context of Fog Computing. A location-aware replica deployment strategy is used to determine the most convenient set of replicas for each data fragment to minimize access latency, creating *Context-of-Interest* (CoI) areas. Consistency within a CoI is ensured by enforcing that all read and write quorums include a majority of replicas. However, consistency is relaxed for replicas outside the CoI which may result in having more distant users accessing outdated data. In our model we guarantee consistency among all replicas storing each data fragment.

4. The self-configuring geographical-aware state deployment service

The adaptive geographical-aware state deployment service aggregates two components: a geo-aware state deployment algorithm, with a scalable distributed environment framework that supports data consistency and replication.

4.1. Geo-aware state deployment

In respect to the algorithm addressing the GeoSD problem, it rests on the principle that past behaviour provides a good indication of the future access patterns. Past behaviour is collected in background by a central authority, hereafter named the *oracle*, from transaction logs. Periodically the oracle evaluates the system performance, decides and coordinates state deployment migrations, using one of the following approaches [52]. These approaches fall into different categories depending on the data structures or the information they assess to take their deployment decisions. For instance, *Linear Approaches* rely on intuitive data structures and algorithms that simply relocate state components to the surrogate with the highest utilization count, or to the surrogate hosting the next transaction manipulating that component.

The *Latency Sensitive Approach* (LR), on the other hand, considers latency of committed transactions to be the most trustworthy metric to represent system performance. The oracle maintains, for each state item, an individual listing of the transactions that manipulated it, the surrogate that initiated the transaction and its completion time. Then LR periodically compares the impact of keeping a component in the current location against moving it to a new surrogate. Eq. (1) depicts how T_{Si} , the time estimated to process all transactions of an object, located at the i th surrogate S_i , is determined. The equation represents the amount of time accumulated by the transactions initiated on surrogate S_i that manipulate the object as t_i and N_i the number of transactions it executed using that object. The equation differs for S_1 and S_i ($i > 1$) because it is assumed that the object is hosted at S_1 and therefore results are those that have been observed, in opposition to the remaining surrogates, where T_{Si} is estimated.

$$T_{Si} = \begin{cases} \sum_{n=1}^i t_n, & i = 1 \\ \frac{t_1}{N_1} N_i + \frac{t_i}{N_i} N_1 + \sum_{j \neq 1, i} t_j, & i \neq 1 \end{cases} \quad (1)$$

T_{Si} is calculated by applying the average transaction time observed in the surrogate holding the object (S_1) to the number of transactions performed by the i th surrogate S_i , plus the average transaction time observed at the i th surrogate S_i to the amount of transactions performed by S_1 , plus the amount of time accumulated by the transactions initiated in all remaining surrogates. The objective of the algorithm is to find the i th surrogate S_i with the lowest value T_{Si} to store the object. As objects are more frequently used together, their respective transaction consumption time tends to decrease if stored in the same surrogate. Conversely, objects that are often used together but are kept in different locations tend to have a greater time consumption.

However, from the evaluation performed in [52], the solution that presented the best results combined both a linear approach with a latency sensitive approach, resulting in the Polling and Latency Combination (P+L). Recall from Section 2 that application state is assumed to be partitioned in 3 categories: *global* state is the information that is relevant to all participants in the system, *location-dependent* state refers to data which is relevant to the participants located in a specific area, and *nomadic* state refers to the information specific to the mobile devices that follows some user. The P+L algorithm, depicted in Algorithm 1, exploits these different characteristics of the application state by relying on the application developer to annotate the state items that fall into each category. For *location-dependent* data the polling approach of counting, for each object, the highest utilized location showed to be enough to define the better suited surrogate. In opposition, *nomadic*

data is evaluated by using the LR approach to compare the impact of keeping the object in the current surrogate with the estimated gains of moving it to a new surrogate. By combining both approaches, the P+L algorithm encourages *location-dependent* objects to be maintained in their most suited surrogate, while focusing on the deployment of *nomadic* data, converging to a better deployment.

Algorithm 1 Polling and Latency Combination State Deployment Algorithm

```

1: Implements:
2:   PollingAndLatencyDeploymentAlgorithm, instance pl.
3:
4: Uses:
5:   BestEffortBroadcast, instance beb;
6:
7: //Initialization.
8: upon event <pl, Init> do
9:   appObjects := appLocations :=  $\emptyset$ ;
10:  locationAmount := locationTime :=  $\emptyset$ ;
11:  totalTransactions := 0;
12:
13: //Register a new object and where it was used.
14: upon event <pl, RegisterObj|obj, loc> do
15:   if loc not in appLocations then
16:     add loc to appLocations;
17:   obj.Location = loc;
18:   add obj to appObjects;
19:
20: //Register a new transaction, containing the set of objects used, the
   issuing location and the commit time.
21: upon event <pl, RecordTransaction|objs, loc, time> do
22:   totalTransactions++;
23:   forall obj  $\in$  objs do
24:     locationAmountloc,obj++;
25:     locationTimeloc,obj := locationTimeloc,obj + time;
26:
27: //Number of transactions registered has reached the predetermined
   number to trigger a new evaluation.
28: upon totalTransactions = EVAL_WINDOW do
29:   trigger <pl, AnalyseDeployment>;
30:
31: //Analysing the deployment of each existing object.
32: upon event <pl, AnalyseDeployment> do
33:   forall obj  $\in$  appObjects do
34:     bestLocation := obj.Location;
35:     sumCurrentTime =  $\sum_{n=obj.Location}^{appLocations} locationTime_{n,obj}$ 
36:     forall currentLocation  $\in$  appLocations where currentLocation
        $\neq$  obj.Location do
37:       if obj.Type = 'geo-dependent' then
38:         bestLocation := max locationAmount
           ((bestLocation, obj), (currentLocation, obj)).Location;
39:       else if obj.Type = 'mobile' then
40:         newEstimation :=
           <pl, TimeEstimation|currentLocation>;
41:         if newEstimation < sumCurrentTime then
42:           sumCurrentTime := newEstimation;
43:           bestLocation := currentLocation;
44:
45: upon event <pl, TimeEstimation|currentLoc> do
46:   return Equation 1(currentLoc);

```

4.2. Lightweight Group view-synchronous communication

The gains observed with the mobility of state deployment motivate the need for providing applications with a tool that support data

Table 2

Lightweight group view synchronous communication specification.

Module	
Name:	Lightweight Group View Synchronous Communication, instance lwgvs
Uses:	ViewSynchronousCommunication, instance vs
Properties:	
LWGVS1 - LWGVS5	same as VS1 to VS5.
LWGVS6 (View Containment)	Let $V_{hwg} = (id_{hwg}, M_{hwg})$ and $V'_{hwg} = (id_{hwg} + 1, M'_{hwg})$ be any two consecutive views delivered by <i>vs</i> . For any view $V_{lwg} = (id_{lwg}, M_{lwg})$ installed by <i>lwgvs</i> after V_{hwg} is delivered and before V'_{hwg} , $M_{lwg} \subseteq M_{hwg}$.
LWGVS7 (Monotonicity)	If a process <i>p</i> installs a view $V_{lwg} = (id, M)$ and subsequently installs a view $V'_{lwg} = (id', M')$, then $id < id'$.

consistency. The *distributed environment* problem aims at providing consistency in a scenario where state fragments may be widespread by surrogates deployed over an unreliable network and also where each application operation may require the participation of multiple surrogates. This is the case expected when the P+L approach for GeoSD is used. This section presents the framework to support the distributed environment of our geographical-aware state deployment service, providing application state consistency in the presence of state partitioning, surrogate crashes and network failures.

Consistency is supported by virtual synchrony, using the LightWeight Group (LWG) [53] abstraction to increase scalability and performance. LWG extends the group view synchronous communication paradigm (HWG) with fine grained, application defined, subgroups without compromising the properties provided by HWG. As depicted in Fig. 2, our implementation of LWG rests on top of two building blocks. A Best Effort Broadcast (BEB) block provides the fast (although unreliable) dissemination of each LWG message. The HWG enforces the reliable delivery of messages and supports membership management. This section begins by formally defining the LWG abstraction and its interface. It then proceeds to describe an algorithm implementing it. The notation in this section refers to the LightWeight Group communication service as LWG, while the set of lightweight group instances an HWG member *m* is affiliated is denoted as $S_m = \{l_1, l_2, l_3, \dots\}$. l_i is used to refer to a single group instance from the previous set.

4.2.1. Specification

The properties of the LightWeight Group (LWG) View-Synchronous Communication abstraction are inherited from the original view-synchronous group communication (HWG) properties (Table 1) and extended with two additional properties, depicted in Table 2.

The first aspect that needs to be ensured is that lightweight group instances are associated to the underlying HWG as some services are delegated to the latter (i.e. group membership). The *View Containment* property establishes this link between the lightweight and the underlying heavyweight views by enforcing the membership of each $l_i \in S_m$ to be a subset of the HWG membership. This property ensures a consistent membership across lightweight groups, avoiding problematic scenarios where the same process may have been declared as faulty on an incomplete subset of the lightweight groups it participates. The second property, *Monotonicity*, enforces a monotonic increase of view ids to facilitate the implementation of recovery procedures. It should be noted that surrogate state recovery and synchronization is application dependent and left to the application programmer.

4.2.2. LWG API

Applications interact with the LWG service using the interface depicted in Table 3. The *l* parameter associates operations to one of the (possibly several) existing groups in S_m . The API presents two

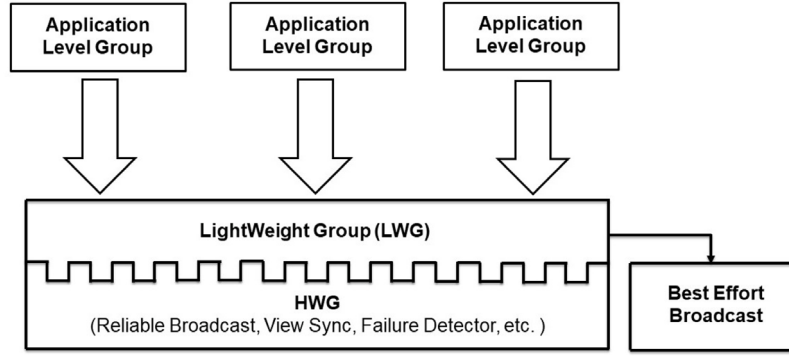


Fig. 2. Two-tier group membership model.

Table 3
Service API of the LWG abstraction.

Operation	Direction	Description
Join(<i>l</i>)	App → LWG	Request to join group with name <i>l</i> .
Leave(<i>l</i>)	App → LWG	Request to leave group with name <i>l</i> .
Block(<i>l</i>)	LWG → App	Indication to stop broadcast new messages in group <i>l</i> until a View or Unblock is received.
BlockOk(<i>l</i>)	App → LWG	Confirmation that no message will be broadcast in group <i>l</i> .
View(<i>l</i>)	LWG → App	Indication of a new view installed for group <i>l</i> .
Unblock(<i>l</i>)	LWG → App	Indication that new messages can be broadcast again in group <i>l</i> .
Send(<i>m</i> , <i>l</i>)	App → LWG	Sends a message <i>m</i> within group <i>l</i> .
Receive(<i>m</i> , <i>p</i> , <i>l</i>)	LWG → App	Delivers a message <i>m</i> broadcast by process <i>p</i> in group <i>l</i> .

categories of operations: group membership management and message exchange. The latter is composed of the *Send* and *Receive* operations, which are associated with a message *m* to be broadcast to the lightweight group. Messages can only be exchanged once a process is a member of a group *l* and a view is installed.

To understand the interaction between applications and the LWG service, Fig. 3 presents the evolution of states applications experience and the operations they use. At the beginning (from left to right in Fig. 3), an application starts by requesting to join a group with the *Join* operation. Eventually, a confirmation is received with a *View*, containing the group membership. At this point, the application can start broadcasting messages to other members. The application developer may use the delivery of a *View* to start a group state synchronization (*State Sync*) and maintain application consistency. Once the synchronization is concluded, the application proceeds to the *In View* state until a *Block* indication is received.

Block signals the need to stop broadcasting messages for the (possible) installation of a new view. The application should move to a *Flush* state where it is allowed to send messages to conclude ongoing operations. Once the application acknowledges this indication with a *BlockOk*, it enters the *Blocked* state where it can still receive the messages sent from the *flush* procedure of other members but is no longer allowed to reply. The *Blocked* state ends upon the installation of a new *View* or with the reception of an *Unblock* indication. The difference between these two indications is whether the LWG service performed any membership changes. If yes, then the application returns to the *State Sync* state. Otherwise, it resumes broadcasting messages at the *In View* state. This *Unblock* indication is unique to our LWG implementation and allows message broadcasting to be resumed, bypassing the more resource consuming operation of group state synchronization.

4.2.3. LWG service in the forest surveillance application

Considering the large-scale forest surveillance system, described in Section 2, lightweight groups would be formed between Ground Stations (GS) depending on the application state they store and manipulate. Recall the example mentioned of a GS requesting more devices to help estimate the extension and progression speed of an existing fire. In such case, multiple lightweight group instances would be created to replicate and validate transactions manipulating the different application state components used in this fire scenario. First of all, a lwg instance would be created including all GS in the region where the fire was first detected, to monitor the latter based on the sensor readings collected in the region. As the fire may progress to adjacent regions, additional GS may have to join the lwg instance to replicate and manipulate the information collected, as well as add new information collected on their regions. In such case, new readings are halted until a new view is installed including the new member(s) and, as they join in, synchronize the current state of the fire with the members already present in the previous view. At that point, all GS are once again in condition to report their measuring and participate in the new transactions. This allows all participating GS to have a consistent view of the fire's current status. Secondly, the need to coordinate the information concerning UAVs or even the status of humans forest guards must be replicated among GS along their path (which may include GS outside the fire area as they are called in), creating new lwg instances to calculate new movement paths or provide new instructions to help fight the fire. In this case, as UAVs or other mobile devices are converging to the fire area, the initial GS that dispatched these devices may voluntarily leave the lwg instance as those application state components are no longer relevant to that GS. Finally, the probability that fire actually destroys a GS or creates network partitions requires that the LWG service must be fault tolerant and allow the application to remain available. The details of how consistency is maintained in the presence of failures and how members of the LWG service coordinate with each other is entailed in the next section by depicting the service's implementation.

4.2.4. Implementation

From the implementation perspective, a single instance of each of the LWG, HWG and BEB building blocks services (depicted in Fig. 2) is running on each surrogate. The LWG service uses the reliable broadcast and membership service of HWG to provide the corresponding properties to the lightweight groups and to ensure state consistency between LWG instances of each surrogate. BEB is used to accelerate application message delivery. Algorithm 2 presents the pseudo-code for the implementation of our lightweight group view-synchronous service.

Algorithm 2 Lightweight Group View-Synchronous Algorithm

- 1: *Implements:*
- 2: LightweightGroupViewSynchronousCommunication, **instance** lwgvs.

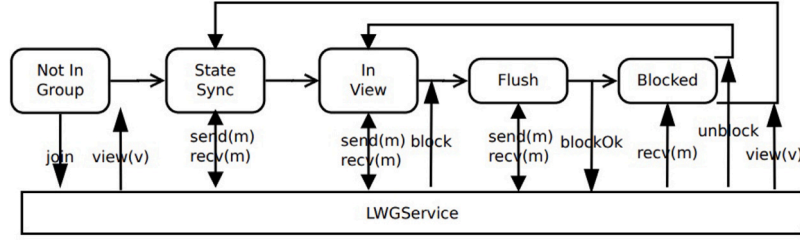


Fig. 3. Evolution of application states when interacting with the LWG service.

```

3:
4: Uses:
5:   ViewSynchronousCommunication, instance vs;
6:   BestEffortBroadcast, instance beb;
7:
8: //Initialization.
9: upon event  $\langle lwgvs, Init \rangle$  do
10:   hwgView :=  $\perp$ ;
11:   vsBlock := TRUE;
12:   blockOk := existingLwgs :=  $\emptyset$ ;
13:
14: //New view delivered to the hwg. Lwgs that keep all members from the
previous hwg view may unblock. The others: mark missing members as
failed processes.
15: upon event  $\langle vs, View[V] \rangle$  do
16:   hwgView := V;
17:   vsBlock := FALSE;
18:   forall  $l \in existingLwgs$  do
19:     if  $V_l.Members \cap hwgView.Members = V_l.Members$  then
20:       blockedl := FALSE;
21:       trigger  $\langle lwgvs, Unblock[l] \rangle$ ;
22:     else
23:       add to failprocsl list:  $V_l.Members$  not in
       hwgView.Members;
24:
25: //Application wants to join lwg group l.
26: upon event  $\langle lwgvs, Join[l] \rangle$  such that  $hwgView \neq \perp \wedge self \in$ 
hwgView.Members do
27:   add to pendingJoin list: group l;
28:
29: //There is at least one lwg group the application wants to join, broad-
casting my requests.
30: upon pendingJoin  $\neq \emptyset \wedge vsBlock = FALSE$  do
31:   forall  $l \in pendingJoin$  do
32:     trigger  $\langle vs, Broadcast[Join, l] \rangle$ ;
33:   pendingJoin :=  $\emptyset$ ;
34:
35: //Some process p wants to join lwg group l.
36: upon event  $\langle vs, Deliver[p, Join, l] \rangle$  do
37:   if  $l \notin existingLwgs$  then
38:     add to existingLwgs list: l;
39:      $V_l := (0, \perp)$ ;
40:     add to joiningProcsl list: p;
41:
42: //The application wants to leave lwg group l.
43: upon event  $\langle lwgvs, Leave[l] \rangle$  such that  $self \in V_l.Members$  do
44:   add to pendingLeavel list: group l;
45:
46: //There is at least one lwg group the application wants to leave,
broadcasting my requests.
47: upon pendingLeave  $\neq \emptyset \wedge vsBlock = FALSE$  do
48:   forall  $l \in pendingLeave$  do
49:     trigger  $\langle vs, Broadcast[Leave, l] \rangle$ ;
50:   pendingLeave :=  $\emptyset$ ;
51:
52: //Some process p wants to leave lwg group l.
53: upon event  $\langle vs, Deliver[p, Leave, l] \rangle$  do
54:   add to failprocsl list: p;
55:
56: //If I am the leader of hwg, hwg is not blocked, there are no pending
views to be installed for lwg group l and I know at least one process
wants to join/leave, then I am proposing a new view for l.
57: upon  $iAmLeader \wedge vsBlock = FALSE \wedge pendingView_l =$ 
 $\perp \wedge (joiningProcs_l \neq \emptyset \vee failprocs_l \neq \emptyset)$  do
58:   pendingViewl =  $(V_l.id + 1, V_l.Members \cup joiningProcs_l \setminus failprocs_l)$ ;
59:   joiningProcsl := failprocsl :=  $\emptyset$ ;
60:   trigger  $\langle vs, Broadcast[NewView, l, pendingView_l] \rangle$ ;
61:
62: //Received a new view proposal for lwg group l. If I am a member, I
must block.
63: upon event  $\langle vs, Deliver[p, NewView, l, V] \rangle$  do
64:   pendingViewl := V;
65:   if  $self \in V_l.Members$  then
66:     trigger  $\langle lwgvs, Block[l] \rangle$ ;
67:
68: //Block for the hwg received. I must block all lwgs where I am a
member.
69: upon event  $\langle vs, Block[l] \rangle$  do
70:   blockOk :=  $\emptyset$ ;
71:   forall  $l \in existingLwgs \wedge self \in V_l.Members$  do
72:     trigger  $\langle lwgvs, Block[l] \rangle$ ;
73:
74: //Lwg group l has confirmed the block. I am flushing the messages
exchanged in this view of l. If this is was the last lwg I was expecting
confirmation, then I can confirm my blockOk to all members via hwg.
75: upon event  $\langle lwgvs, BlockOK[l] \rangle$  do
76:   blockedl := TRUE;
77:   trigger  $\langle vs, Broadcast[Flush, l, V_l, inview_l] \rangle$ ;
78:   blockOk := blockOk  $\cup \{l\}$ ;
79:   if vsBlock = FALSE then
80:     if  $\{l \in existingLwgs : self \in V_l.M\} \subseteq blockOk$  then
81:       trigger  $\langle vs, BlockOk[l] \rangle$ ;
82:       vsBlock := TRUE;
83:
84: //Flush received from p for lwg group l in the same view. Deliver to the
application all missing messages.
85: upon event  $\langle vs, Deliver[p, Flush, l, V, vmgs] \rangle$  such that  $V_l.id =$ 
 $V.id$  do
86:   flushrcvdl := flushrcvdl  $\cup \{p\}$ ;
87:   if  $self \in V_l$  then
88:     forall  $(s, m) \in vmgs$  such that  $m \notin delivered_l$  do
89:       deliveredl := deliveredl  $\cup \{m\}$ ;
90:       trigger  $\langle lwgvs, Deliver[l, s, m] \rangle$ ;
91:
92: //Whenever I broadcast an application message via BEB, I store the
message for the flush procedure and deliver the message to myself.
93: upon event  $\langle lwgvs, Broadcast[l, m] \rangle$  such that blockedl = FALSE do
94:   inviewl := inviewl  $\cup \{self, m\}$ ;

```

```

95:   deliveredl := deliveredl ∪ {m};
96:   trigger ⟨lwgvs, Deliver|l, self, m⟩;
97:   trigger ⟨beb, Broadcast|Vl.M, [Data, l, Vl.id, m]⟩;
98:
99: //Message received from p to lwg group l. Deliver to my application and
   store for the flush procedure.
100: upon event ⟨beb, Deliver|p, [Data, l, v, m]⟩ such that v = Vl.id do
101:   if m ∉ deliveredl then
102:     inviewl := inviewl ∪ {(p, m)};
103:     deliveredl := deliveredl ∪ {m};
104:     trigger ⟨lwgvs, Deliver|l, p, m⟩;
105:
106: //If a new view for lwg l has been proposed, I have received all flush
   messages from the members of l that will remain from the previous view
   to the next, l is blocked and the hwg is in view, then I can install the
   new view for l.
107: upon pendViewl ≠ ∅ ∧ Vl.M ∩ pendViewl.M ⊆ flushrcvdl ∧
   blockedl = TRUE ∧ vsBlock = FALSE do
108:   Vl := pendViewl;
109:   pendViewl := ⊥;
110:   inviewl := flushrcvdl := ∅;
111:   joinprocsl := joinprocsl \ Vl.M;
112:   failprocsl := failprocsl ∩ Vl.M;
113:   if self ∈ Vl.M then
114:     trigger ⟨lwgvs, View|l, Vl⟩;

```

View management. To make the service oblivious to process failures, the service replicates views state (composed by the order list of members and view id) of every l_i by all members of the heavyweight group. Views state is kept consistent across all replicas by using the HWG reliable broadcast to disseminate all *Join* and *Leave* requests issued by every process. The installation of lightweight group views is also coordinated via HWG reliable broadcast messages.

There are three situations that can lead to the installation of new views: (a) a request to join is received, (b) a request to leave is received or (c) when any member of the HWG is suspected to have failed. In the latter, it should be noted that until the new heavyweight group view is installed, it is not possible for the LWG service to learn which (if any) lightweight groups are affected. The LWG service propagates the *Block* indication received from the HWG to every managed lightweight group. As soon as the new heavyweight group view is installed, the HWG leader compares the membership of every lightweight group with the list of heavyweight group members. For those lightweight groups which have seen their membership changed, the leader triggers the installation of a new view. For those whose heavyweight group membership change had no impact on the lightweight group view, the leader coordinates the resume of the view, which is converted by every instance of the LWG service on an *Unblock* indication. Coordination is once again assured by the use of the HWG's reliable broadcast service to disseminate the leader's decisions. Table 4 summarizes the impact of membership changes on each l_i and the interaction between the HWG and LWG communication services. The *Unblock* primitive limits the impact on lightweight groups of HWG view changes. Only those groups whose membership needs to change are affected, while the remaining can resume message broadcasting.

A particular case of view management happens when the leader of the heavyweight group fails. The problem is solved considering 2 factors: i) the HWG service elects a new leader with every view; and ii) the use of the reliable broadcast service ensures that the view state of every lightweight group is consistent and can be synchronized with joining processes. Hence, the new HWG leader can perform the coordination tasks as soon as the new heavyweight group view is installed.

Table 4

Impact of membership changes on view installation.

Incident	View change	
	HWG	LWG
p joins HWG	•	
p suspected	•	◦
p joins lwg l_i		l_i
p leaves lwg l_i		l_i

•: View change; ◦: View change on some groups in S_l ;
 l_i : View change on group l_i .

4.2.5. Message broadcasting and consistency

Application data messages issued by a process in the scope of l_i are disseminated by the LWG service using the BEB service. The BEB service uses a set of TCP connections to implement a low-cost message delivery service where messages are exclusively delivered to the processes affiliated with l_i . Although the BEB service scales linearly with the number of group members, it does not ensure the reliable broadcast properties, which are expected from a service providing virtual synchrony such as LWG. In particular, we are concerned with the property stating that if some correct process delivered a message m then all the correct processes must deliver m . To enforce reliable broadcast properties, the LWG service running on surrogate S adds every message either broadcast or delivered to lightweight group l_i to a set M_{m,l_i} .

4.2.6. Reliable delivery enforcement

A lightweight or heavyweight group is considered blocked when processes are not allowed to send messages. Blocking is requested with the *Block* indication and begins when processes reply with a *BlockOK*. This confirmation is important as it indicates that no further messages will be sent in the scope of the view and, from the surrogate's perspective, M_{m,l_i} is closed. Once the group is blocked, proceedings to ensure reliable broadcast properties are as follows. At the lightweight group level, the LWG service uses the reliable broadcast primitive of the HWG to disseminate the content of set M_{m,l_i} for every group l_i . The properties of reliable broadcast ensure that the union of M_{m,l_i} sets received from every correct member of the lightweight group contains all messages delivered to any correct member of the view. The LWG service at every surrogate can then compare its lists of messages with the unions received and deliver to the application any message not delivered by BEB.

Reliable delivery is thus only enforced when a view changes is triggered. Fig. 4 illustrates the steps followed in a view change and the flow of messages exchanged. All cases that trigger a possible view change are initially handled equally, with the particularity that a *Block* at the heavyweight group (1) triggers a *Block* on every lightweight group the LWG service is a member (2). In the case the new view is triggered by a change in the heavyweight group membership, the sets M_{m,l_i} are reliably broadcasted (4) before the *BlockOK* is forwarded by the LWG service to the HWG service (5). The list of messages is computed after the new view is installed (7,8). The installation of the new view or the *Unblock* of the lightweight group immediately follows (9).

The algorithm above is an extension of the *flush* procedure defined in [54] to encompass lightweight groups. It is applied on every view change either at the lightweight or heavyweight group level with the additional gain of postponing the impact of reliable broadcast to the moment where view changes occur. Considering the large ratio between the number of messages and the number of views, this is expected to significantly contribute to improve performance without sacrificing reliability. It should be noted that changes in the heavyweight group membership can result in the execution of the flush algorithm without a view change. This is the case when the view blocks and then unblocks, as no change in the lightweight group membership was found.

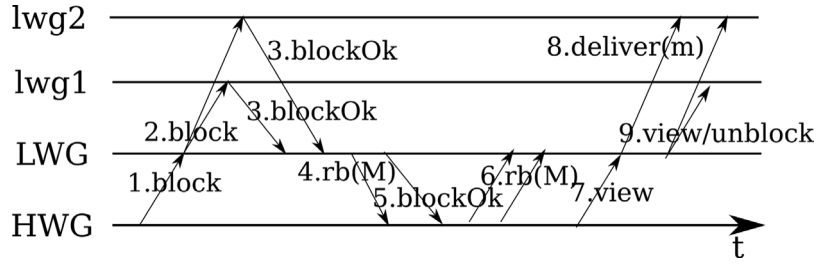


Fig. 4. Steps in installation of a new view at the heavyweight group.

4.2.7. Correctness

The algorithm's proof of correctness starts by considering the *view inclusion* and *agreement* properties. Let m be any message sent by surrogate p in view V_{l_i} of lightweight group l_i and delivered by some correct surrogate q . There are two alternatives for the delivery of m . In the general case, addressed in Section 4.2.5, member q delivers m after the LWG service received it from the BEB service. The case where the BEB service fails to deliver m is addressed in Section 4.2.6. q delivers it in the context of the flush procedure by some correct surrogate r that delivered m . Recall that the flush procedure occurs in anticipation to the installation of the next view and that it relies on the reliable broadcast provided by the HWG service.

The *validity* property is implemented at the LWG service by triggering the delivery of message m in response to every broadcast of m received from p .

The *no duplication* property follows from the application of the M_{p,l_i} set for storing every message m delivered to surrogate p in the scope of lightweight group l_i and from a verification $m \notin M_{p,l_i}$ prior to forwarding the delivery indication.

The *no creation* property follows from the properties of the best-effort broadcast and heavyweight group services.

The use of the reliable broadcast provided by the HWG service to manage view changes is key for the satisfaction of the *View Containment* property. Recall from Property VS1 that a surrogate p can join a new lightweight group l_i with view V_{l_i} if an underlying heavyweight group h exists and p is a member of V_h . Assume there is a moment where p belongs to some lightweight group l_i with view V_{l_i} , but $p \notin V_h$. Such assumption would imply that, at some point, a view change excluding p would have been triggered by h . The installation of the new view V'_h , excluding p necessarily forces all lightweight groups where p is a member to trigger a view change and install a new view V' excluding p . Hence, by contradiction, it is impossible to achieve an execution where $p \in V_{l_i}$, but $p \notin V_h$.

The *monotonicity* property follows directly from the increment of the view id, performed by the LWG service, whenever a new view is installed.

5. Evaluation

To evaluate the scalability and performance of the geo-aware state deployment service presented in this paper, comparable versions were implemented and evaluated using the virtual synchrony communication abstraction or the ZooKeeper [17] reliable distributed coordination service. The goal is to understand if the service can attenuate the well known scalability and performance limitations attributed to virtual synchrony and understand how does it fare with ZooKeeper, a reference in coordination services that is supported and used by a large number of Cloud-based applications.

In total, four approaches were evaluated.² The baseline virtual synchrony setting (hereafter referred as *hwg*) uses a single virtual synchrony group on the View-Synchronous Group Communication service *Appia* [44], having all surrogates as members. The *lwg* approach results from the implementation of the LWG service presented in this paper, resting on top of the off-the-shelf *Appia* implementation. Each surrogate participates on the service as an HWG member, on top of which fine grained lightweight groups are created by the LWG service. The contribution of the oracle is evaluated by running a comparable version of *lwg*, hereafter named the *lwg-Oracle* approach, which combines the lightweight groups of the LWG service with the geo-state deployment P+L algorithm that periodically remaps the current dataset deployment. It should be noted that migrations are performed by the installation of a new view for that object's *lwg* group, with the new members responsible for storing it and excluding of the previous ones. Finally, the *Zookeeper* approach is implemented with every surrogate having a local zookeeper instance to which they communicate and issue their application transactions. The Zookeeper tree is responsible to store every object in the dataset and notify the respective surrogate when its respective transactions have been successfully committed. One important aspect of Zookeeper is that, internally, each write request is forwarded to the leader znode instance which then atomic broadcasts the changes to all follower znodes. This means that zookeeper provides strong consistency by default, which is not the case of the HWG and LWG services as they only guarantee a reliable communication primitive. Hence, to level all approaches to a common strong consistency ground, in the *hwg*, *lwg*, and *lwg-Oracle* approaches, transaction validation is implemented via a voting scheme where the transaction issuer only commits the transaction when it has received a voting confirmation from all other members that need to validate it.

We start our evaluation by comparing the *lwg* and *lwg-Oracle* approaches with the coarse grained *hwg* on top of which they rest. To compare the performance of these approaches the following metrics are assessed. The **transaction duration** measures the time elapsed between a new transaction request from a surrogate and the moment it was able to successfully commit it, averaged by the number of transactions issued by the surrogate during one test. Transaction duration includes sending the request to all members that need to validate the transaction and gather their respective votes to commit. The **number of transactions** metric highlights the average number of transactions that each surrogate validates throughout the test and assesses the level of scalability each approach offers. Smaller number of transactions received by each surrogate means that transactions need the participation of fewer members, hence more transactions are expected to be committed in parallel. The **percentage distribution of transactions per number of members involved** allows the comparison between the *lwg* and *lwg-Oracle* approaches in respect of data deployment,

² The experiments described in this section, the *lwg* implementation as well as the data used in the tests will be made publicly available on a git repository. The location will be included here on the camera-ready version of the paper. The *Appia* protocol composition framework is publicly available at the framework website on <http://appia.di.fc.ul.pt>.

Table 5

Types of transactions and respective percentages on both workloads evaluated.

Transaction type	Workload	
	Small groups (SG)	Large groups (LG)
2 State components	90%	5%
3 State components	5%	90%
Variable number	5%	5%
Total transactions	200 000	200 000

assessing the impact and benefits of using a GeoSD algorithm when compared to static and random state deployment as followed in the *lwg* approach. It is expected that, with the *lwg-Oracle* approach, the majority of transactions will count with the participation of fewer members as it aims to a deployment where all the objects are hosted in the same surrogate prior to a transaction.

5.1. Workload characterization

To emulate a Fog based mobile distributed application, two transaction workloads were generated from the Austin's b-cycle³ application public dataset. Austin b-cycle is a bicycle sharing service from the city of Austin, USA, where users can checkout and return bicycles from specific stations (called *kiosks*) deployed around the city. The dataset includes both the kiosks location and the trips performed by b-cycle users.

Three transaction types were created from the dataset, depending on the number of application state components manipulated. The first transaction type consists on either checking out or returning a bicycle to a kiosk, thus involving two state components (a bicycle and a kiosk). The second transaction type checks out a bicycle from a kiosk and reserves a parking slot at the destination, involving 3 application state components. The third transaction type involves a variable number of state components as it probes kiosks on a 500 m radius for an available bicycle and make the respective checkout. Two testing workloads were then devised, having different percentages of transactions from each type as depicted in Table 5. The baseline testing workload, i.e. *Small Groups (SG)*, which favours transactions having less application state components and less participants per transaction, is expected to highlight the benefits of the *lwg*-based approaches. The second workload, hereafter named *Large Groups (LG)*, is composed of the opposing ratio of transaction distribution, representing a more challenging scenario where each transaction needs to be delivered to a larger number of participants. Each workload contains a total of 200 000 transactions.

5.1.1. Application state and surrogate servers

From the application state standpoint, these workloads present both *location-dependent* and *nomadic* state components. The kiosks are fixed delivery and pick-up points of bicycles containing an address, the total number, and the number of available parking slots. Such information is relevant to the participants located in proximity and less accessed by more distant participants, representing location-dependent data. On the other hand, each bicycle information (containing its ID, whether it is in use and the respective trip trajectory) is relevant to the user interacting with it and represent nomadic state.

Each of these application state components is stored in the surrogate servers closer the kiosk's address or to where the bicycle was first used. During evaluation, the city map of Austin is split according to the number of surrogate servers tested, so that each set of 3 surrogates is responsible for an even sized region of the city. This replication factor of 3 ensures availability and, together, these surrogate servers create the component's *lwg* and are responsible for validating each transaction that manipulates that specific state item. For the transactions that manipulate multiple state components, all surrogate servers that store at least one are grouped to form a larger *lwg* that validates the transaction.

5.1.2. Testbed configuration

Scalability is evaluated running the experiments with 6 to 18 surrogates. To stress the geographical distance between surrogates, surrogates were deployed on different points of the European continent. Each surrogate runs on a *t2.medium* Amazon EC2 instance on one of 5 Amazon datacenters located in Ireland, London, Milan, Paris and Stockholm. The evaluation consists on running all transactions of a workload with the four proposed approaches. In the *lwg*-based approaches, the surrogates defining each lightweight group replicating an object are co-located in the same datacenter. The locations of each application state item is known a priori by all surrogates and remain static. The only exception is the *lwg-Oracle* approach, where the geo-state deployment algorithm is running in a separate EC2 instance that functions as the Oracle, responsible for collecting the transactions logs sent the by surrogates, and starts a new deployment evaluation at each 10 000 transactions logs received.

On average, each transaction carries 40 bytes of raw, non-compressed data. A test consists on having each surrogate issuing the 200 000 transactions. In the *hwg* approach, transactions are exchanged in the single core group and therefore must be validated by every member. It should be noted that the performance of the system is randomly affected by the latency observed at each moment between Amazon datacenters. To reduce the bias, the runs of each approach were executed in sequence. Results presented in the paper are the average values of 10 runs in comparable conditions.

5.2. Evaluation results

5.2.1. High pace broadcast test

In this first test a highly loaded system is simulated where each surrogate broadcast a new transaction at every 5 ms. Expectations are that, by broadcasting transactions to different subgroups in parallel, the *lwg*'s scalability will be evidenced and better (i.e. faster) results are expected in comparison with the baseline *hwg* single group approach. As the system size increases, the *lwg-Oracle* approach is expected to outperform even further the linear *lwg* approach as it will be able to cope with an increasing dataset fragmentation and random deployment, by being able to remap state deployment in order to reduce transaction duration.

From the users' standpoint, the most important metric is to assess how fast their transactions commit. Moreover, as latency is a difficult metric to assess individually and may be influenced by multiple factors such as server load to process events or the varying network congestion found during execution, we consider that latency is indirectly assessed via the average transaction duration. In face of smaller transaction durations, then latency has been reduced as well. Fig. 5 depicts the average transaction duration obtained by *hwg*, *lwg* and *lwg-Oracle* approaches. The first observation that stands out is that transactions' commit time is in the order of seconds when the average round-trip time between Cloud datacenters nowadays varies in the order of hundreds of milliseconds. To explain such high results, we need to recall the test configuration and how transactions are validated. In the test, each surrogates issues around 200 000 transaction at a 5 ms pace, which means that within seconds of a simulation each surrogate can have up to 20 000 events to process with just the smallest system size of 6 surrogates. Added to these events are all the others related with group membership management, which compete for the same resources, like the heartbeat messages destined to the failure detector. These results evidence the poor performance of the virtual synchrony frameworks in general and the *Appia* framework used in these simulations. In particular, when faced with an excessive load in the system, these frameworks may even lead to erroneous failure suspicions due to the delay in responding to heartbeat messages. However, as the conditions remain equal for the three approaches, we can still perform a fair comparison among them. Thus, it is observable that the *hwg* approach presents a linear growth throughout the tests that is not followed by

³ <https://austin.bcycle.com/>

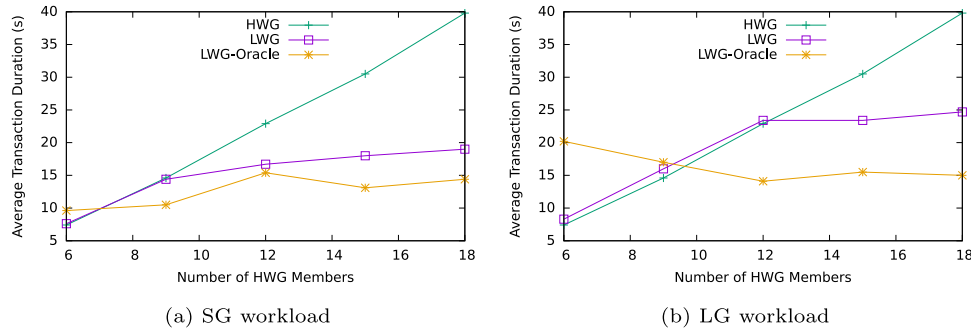


Fig. 5. Average transaction duration in the High Pace Test.

the *lwg*-based approaches. In both workloads, the *lwg-Oracle* approach is the one presenting better (i.e. smaller) transaction duration. In fact, it is outperformed by both *hwg* and *lwg* in both scenarios with the smallest system size of only 6 members, showing that the execution of GeoSD algorithm is being counterproductive to the system. As there are few surrogates and the dataset follows a more uniform utilization pattern, the GeoSD keeps chasing the state components and moving them around surrogates without bringing any benefit to the system. As the system size increases, the dataset becomes more fragmented and the benefits of relying on a GeoSD algorithm become more evident. On the other hand, the *lwg* approach starts with a linear growth just as *hwg* for the SG workload with the system size up to 9 surrogates and with up to 12 surrogates in the LG workload, respectively. Beyond those points, the *lwg* flattens and diverges from *hwg*. The *hwg*'s linear increase is explained by the system size evolution where each surrogate that is added means 200 000 more transactions that need to be delivered and validated by all members. Conversely, the transaction duration's increase in the *lwg* approach flattens its growth as the addition of new members (i.e. more transactions to validate) is attenuated by the creation of additional groups that leads to a more fragmented dataset, where the probability of having to reach all surrogates because each store at least one state component manipulated in a transaction decreases. Nevertheless, the difference between the SG and LG workloads is clear as the *lwg*-based approaches present transaction durations 25% larger in the latter on average. This shows the influence of the workload in system performance where, in a scenario where transactions need to reach a majority of participants, the gains observed by the *lwg* approach is attenuated. On the other hand, the benefits of applying a GeoSD algorithm become more evident as the transaction duration of *lwg-Oracle* in the LG workload allows almost a 50% improvement over the baseline *lwg* approach.

5.2.2. Resource consumption in the high pace broadcast test

To further highlight the impact of transactions on each surrogate, we now focus on the number of transactions received by each member. Expectations are that, by creating different subgroups in, the *lwg*-based approaches approach are able to reduce the amount of transactions received by each surrogate. The benefits of receiving less transactions are twofold. On one hand, note that the reception of a transaction has an impact on CPU utilization to process its messages. But, more importantly, it directly contributes to reduce network traffic and jitter as there are less messages circulating in network, competing for the same bandwidth. Fig. 6 depicts the average (crosses), maximum, minimum and mean (circles) of the number of transactions received by each surrogate. The approaches in this figure show a pattern similar to Fig. 5 with the average transaction duration depending on the amount of transactions each surrogates receives. As expected, the four metrics remain close in the *hwg* approach as every member receives the same amount of transactions and the increase in system size represents a steady addition of 600 000 (200 000 * 3 new members) transactions for every participant to validate. In opposition, the *lwg*-based approaches

present linear growths, at the edge of 1.5 million transactions per member at most. This means that even with a system size of 18 surrogates, a surrogate might have to participate at most in the same amount of transactions than in the *hwg* approach for the smallest system size. Moreover, considering that the amount of transactions received per surrogate in the *lwg*-based approaches varies from a minimum of 1 Million (6 surrogates) to a maximum of 1.5 Million (18 surrogates) on average, and that each transaction carries around 40 bytes of data, to the total amount of network traffic received by each surrogate is $40 \times 1\,500\,000 \approx 60\text{MB}$ at most. In the *hwg* approach, on the other hand, the total number of transactions received per surrogate can reach up to 3.5 Million, all competing for the same bandwidth, translating in a total network traffic of $\approx 133\text{MB}$. Hence, the LWG is able to show its scalability by reducing to the less than half the total amount of transactions exchanged, reducing the probability of jitter, and the total amount of network traffic characterized by amount of data received per surrogate.

Among the *lwg*-based approaches, one can observe that *lwg-Oracle* presents an average number of transactions per member below the average of *lwg* and this difference tends to increase as the system size also increases. On the other hand, the minimum amount of transactions received by a member decreases sharply when compared to the minimum values obtained in the *lwg* approach and the *lwg-Oracle*'s maximum actually surpasses the *lwg* maximum. Thus, contrary to the *lwg* approach that randomly distributes data objects and obtains an uniform distribution between minimum, maximum, average and mean, the *lwg-Oracle* approaches shows the effectiveness of applying a GeoSD algorithm combined with the LWG service. The oracle has the ability to remove misplaced state items from surrogates, making some members to receive less than half of transactions than the best scenario of the *lwg* approach. If a member possesses an object that it does not manipulate according to its transactions workload, the GeoSD will quickly remove that object from that location as it is hampering the system and the surrogate ceases to participate in transaction validation of data that it does not dispute with other members. Conversely, some members that manipulate a larger variety of objects in their transactions, disputing their object deployment with other members, inevitably received more transactions and the maximum values of the *lwg-Oracle* approach above the *lwg*'s show this unbalance. This behaviour is further highlighted with the LG workload. However, this is not considered to be an issue as the GeoSD algorithm is supposed to map as closely as possible the data utilization pattern it observes and not function as load balance. If some members manipulate more objects and if concentrating their deployment in fewer members benefits the system, then the GeoSD will reproduce that behaviour. Nevertheless, as the average and mean of the *lwg-Oracle* approach remain below *lwg*'s and their discrepancy increase with large system sizes, we can conclude that this overload upon some surrogates is the exception and not the standard behaviour.

For completeness in the analysis of the LWG service when combined and not combined with the GeoSD algorithm, Figs. 7 and 8 depict the percentage of transactions based on the number of replicating groups

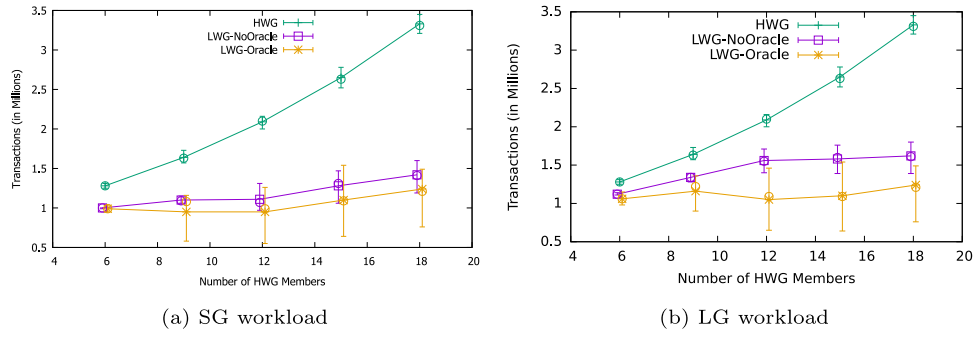


Fig. 6. Average, maximum, minimum, and mean (in circles) of transactions validated per surrogate in the High Pace Test.

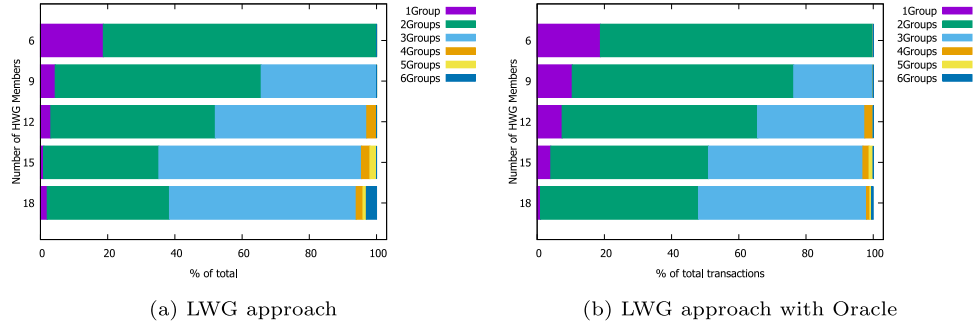


Fig. 7. Percentage distribution of transactions per the replication groups involved in the SG workload.

(each group composed of 3 surrogates) needed to validate a transaction with each workload. The best possible result obtained would be 100% of transactions requesting a single group to validate them, representing the perfect deployment where objects are enclosed in the surrogates that manipulate them with no interference in dataset manipulation from other outside members. Unfortunately, such scenario is highly unlikely and transactions need to rely on the distributed environment. Still, the smaller amount of groups reached per transaction the better. In this sense, aside the smallest system size of 6 surrogates where both *lwg* and *lwg-Oracle* approaches present very similar values in the SG workload, the benefits of applying a GeoSD are clear in all remaining scenarios. This is specially true with the LG workload, where the presence of GeoSD algorithm allows a dynamic state item deployment that benefits the system performance by migrating state items manipulated in conjunction to the same replication groups, contributing to present very similar percentage distribution of transaction per group, even if the initial workload does not favour such behaviour. The percentages depicted in Figs. 7 and 8 thus support the values presented in Fig. 6 where, by nearly avoiding all transactions that need all or a majority of members to participate, some members will benefit of having smaller amounts of transactions to validate. On the other hand, by promoting these “edge” transaction with a majority of members to become transactions that reach only 2 or 3 groups at most, some surrogates become overload with more objects deployed and hence have to participate in a larger number of transactions, resulting in higher maximums discussed before. The only exception in this behaviour is in the smallest system size, where the GeoSD is not able to improve the transaction distribution among members. In fact, since there is such a small number of surrogates, the objects usage remains uniform among all members and the deployment algorithm becomes counter-productive in attempting to obtain a better state deployment. The percentage of transactions remains similar between *lwg* and *lwg-Oracle* approaches, with the added penalty of the latter of forcing state migrations and consequent group view changes with no direct benefits to the system. This explains why, in respect to transaction duration depicted in Fig. 5, the *lwg-Oracle* approach actually presents worse duration than *lwg*.

5.2.3. LWG and ZooKeeper tests

To evaluate the impact of the LWG service, we also compared the previous approaches against ZooKeeper, a benchmark in reliable distributed coordination services, widely used in the industry. Throughout this subsection only the *lwg-Oracle* approaches is depicted in the figures against ZooKeeper as it exhibits the best performance of the approaches studied before. ZooKeeper structure was used to store the objects of the dataset as a consistent distributed database. ZooKeeper is known for providing group membership services and is a good candidate to implement a service similar to LWG. However, as discussed in Section 3, this may result in inconsistencies difficult to solve as one process may already be seen as faulty in some groups while still being considered correct in others. Moreover the blocking ZooKeeper API was used, i.e. every surrogate can only issue a new transaction once the previous has been committed. The alternative would be to use the non-blocking API which would require the implementation of consistency and rollback mechanisms by the application programmer, which is counter-intuitive to our proposal of the LWG service that consists in simplifying application development.

Fig. 9 depicts the average completion of ZooKeeper against the *lwg-Oracle* approach in the high pace broadcast test. In the best case scenario, i.e. the smallest system size, ZooKeeper takes triple the time to complete the workload and continues on increasing that completion time for a total 8x times than *lwg-Oracle* with a system of 18 surrogates. This result shows the impact on transaction execution of ZooKeeper relying on total-order broadcast and on the serialization imposed by the blocking API.

On the other hand, considering the average transaction duration to commit, depicted in Fig. 10, zookeeper presents results an order of magnitude better than the ones showed in Fig. 5, around 20 ms to 80 ms when *hgw*, *lwg* and *lwg-Oracle* were around seconds. To investigate if this advantage of ZooKeeper could be attributed to the small number of transactions running in parallel on ZooKeeper, the *lwg-Oracle* was experimented with a new transaction pace of 500 ms. This *slow pace* test simulates conditions closer to those of ZooKeeper where surrogates are not overwhelmed with a transaction generation pace

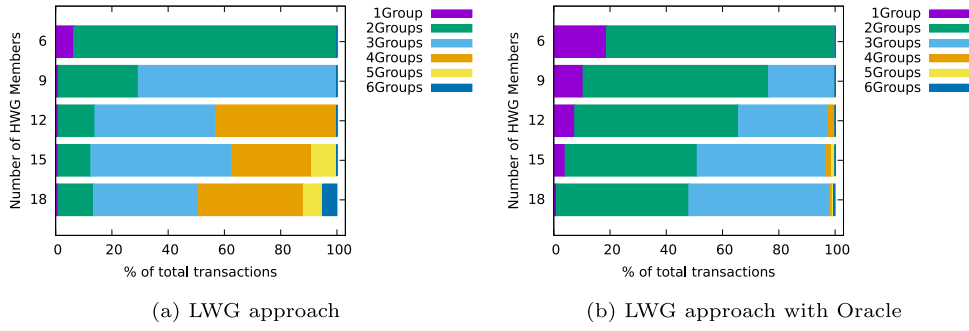


Fig. 8. Percentage distribution of transactions per the replication groups involved in the LG workload.

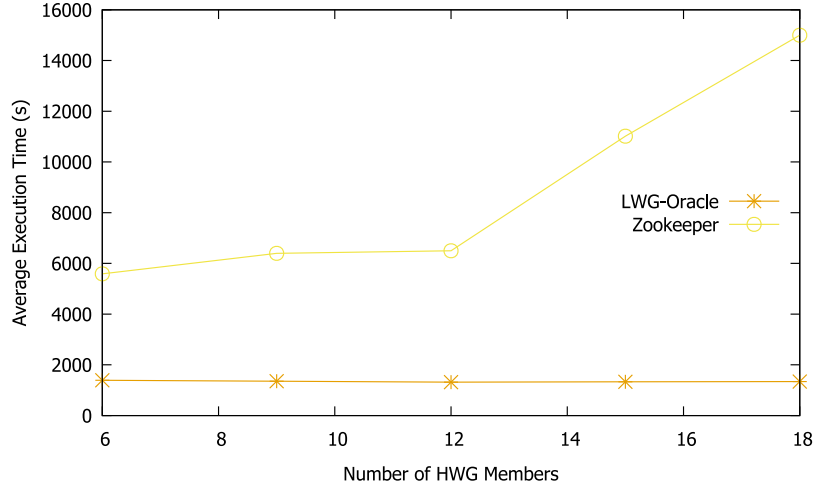


Fig. 9. Average completion time in the SG workload.

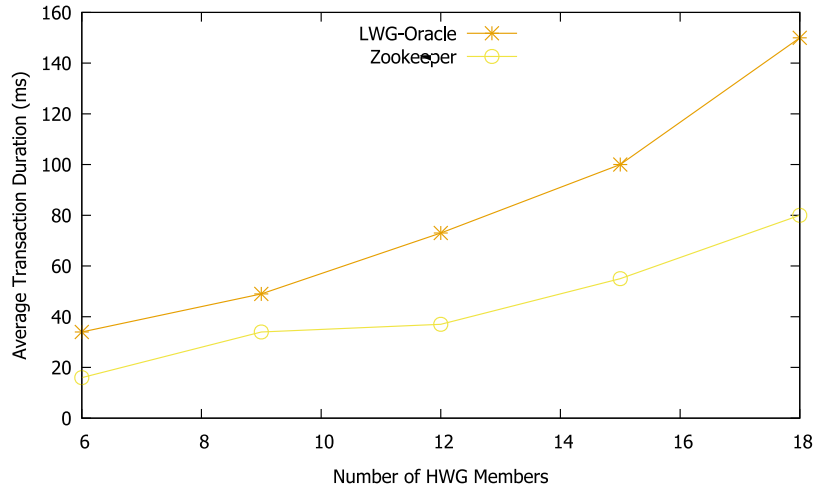


Fig. 10. Average transaction duration in the Low Pace Test and Zookeeper.

they cannot keep up. Results are depicted in Fig. 10. Observing these values one can conclude that, in similar conditions, the transaction duration of *lwg-Oracle* can be brought to the same order of magnitude of ZooKeeper (ms), but each transaction still takes double the time to commit showing the ZooKeeper's efficiency.

5.3. Discussion

Fog Computing has the potential to mitigate jitter, end-to-end latency and reduce load in the backbone networks as long as 2 conditions

are met: (i) a state deployment strategy able to efficiently deploy and relocate data to the surrogates closer to where it is more frequently used and (ii) a process coordination mechanism able to support the emergence of distributed transactions while maintaining scalability.

The evaluation of the LWG abstraction focused on two different aspects: scalability and performance, characterized by the analysis of average transaction duration, number of transactions received and respective network traffic generated, as well as total execution time. Table 6 presents a brief summary of the results obtained by each approach evaluated in each metric. All results refer to the system size

Table 6

Performance results of each approach evaluated with a system size of 18 surrogates in the SG workload.

Approach	Metric			
	Avg duration p/ transaction	# Transactions (Total)	Network traffic	Execution time (Total)
HWG	39.8 s	3.33 Million	127 MB	1155.4 s
LWG	19.0 s	1.42 Million	54 MB	1133.2 s
LWG-Oracle	14.4 s	1.24 Million	47 MB	1342.8 s
Zookeeper	80 ms	1.42 Million	54 MB	15000 s

of 18 surrogates, representing the most challenging configuration in scalability and performance. For each metric, the best result obtained is highlighted in *italic*. One can observe that the *lwg-Oracle* approach presents the best results in 2 metrics (total number of transactions received and network traffic), Zookeeper and *lwg* both have one best result (in average transaction duration and total execution time, respectively), while the *hwg* has none. The fact that *hwg* does not outperform none of the other approaches in any metric is not surprising as it was considered as the baseline test from the beginning.

However, comparing both variants of the LWG abstraction with ZooKeeper is more interesting as we obtain very distinct results. When we consider a faster transaction issuing pace, the *lwg*-based approaches present average transaction duration times incomparable slower than ZooKeeper, but are able to process the entire workload in a matter of minutes instead of hours. To present comparable average transaction durations, the *lwg*-based approaches require almost one transaction to be validated at the time in the system, resulting from a 500 ms interval between each transaction issued. This means that there is a trade-off between the LWG abstraction and ZooKeeper. The former can cope with a greater degree of transaction parallelism, resulting in faster completion times at the cost of slower transaction duration. ZooKeeper is unable to cope with parallelism but present much faster duration to commit transactions. Despite the differences, the most important conclusion is that our approach and ZooKeeper are not competitors, but rather aimed to different purposes and applications. In scenarios such as a the bicycle sharing app or the forest surveillance system, where the checkout of a bicycle or a UAV monitoring do not dependent of concurrent transactions using distinct bicycles or mobile devices, the parallelism and scalability provided by the LWG abstraction should be exploited to improve overall system performance. Otherwise, in situations that require transaction ordering, ZooKeeper is recommended.

Within the LWG abstraction, the objectives pursued by creating multiple subgroups with smaller affiliations have been confirmed: the amount of transactions and network traffic received per surrogate, as well as the respective average transaction duration have been reduced in both *lwg*-based approaches. These results are further improved when the LWG abstraction is combined with a GeoSD algorithm that periodically remaps the current dataset deployment, showing that the geographical relevance of application state components can help improving system performance. Nevertheless, it is important to notice that, through the comparison between the SG and LG workloads, these results and the applicability of a *lwg*-based solution is only possible with applications that do not obligate every transaction to be delivered to every member, such as a bicycle sharing management application, a multi-user augmented reality game, or forest surveillance system.

6. Conclusions

Large scale mobile distributed applications must manage a large number of shared objects (the application state) and provide a consistent view to every participant. Deploying the application state in the Cloud facilitates state management and consistency assurance, but creates a geographical barrier between the end users and the datacenters. Fog Computing architectures can mitigate this impact by deploying

surrogate servers at the network edge, but creates additional challenges that could not be found in the centralized cloud infrastructure. The first is location management, where Fog Computing's performance strongly depends on the ability to deploy each of the application state components at their most convenient location. The second is the distributed environment, characterized by an increasing number of user interactions requiring the participation of multiple surrogates. To address these challenges, this paper proposed a self-configuring geographical-aware state deployment middleware that aggregates a geo-aware state deployment algorithm with a distributed environment framework scalable to a large number of surrogates. The framework tailors virtual synchrony to Fog Computing by extending the original concept with dynamically defined fine grained virtual synchronous groups, while the geo-aware state deployment algorithm exploits different characteristics of the application state to find the most suited location for each, according to their usage pattern. Together, these approaches form a middleware that is scalable as it forwards transactions to a smaller number of participants, saving surrogates resources, improving transactions parallelism and consequently system performance.

CRedit authorship contribution statement

Diogo Lima: Conceptualization, Methodology, Software, Writing – original draft, Writing – review & editing. **Hugo Miranda:** Conceptualization, Methodology, Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgements

This work was supported by FCT, Portugal through the LASIGE Research Unit, ref. UIDB/00408/2020 and the Individual Doctoral Grant, Portugal, ref. SFRH/BD/120631/2016.

References

- [1] D. Reinsel, J. Rydning, J. Gantz, Worldwide Global DataSphere Forecast, 2021–2025: The World Keeps Creating More Data — Now, What Do We Do with It All? IDC, 2021.
- [2] T. Lähderanta, T. Leppänen, L. Ruha, L. Lovén, E. Harjula, M. Ylianttila, J. Riekk, M.J. Sillanpää, Edge computing server placement with capacitated location allocation, J. Parallel Distrib. Comput. 153 (2021) 130–149.
- [3] M. Gerla, Vehicular cloud computing, in: 2012 the 11th Annual Mediterranean Ad Hoc Networking Workshop, Med-Hoc-Net, 2012.
- [4] S. Agarwal, J. Dunagan, N. Jain, S. Saroui, A. Wolman, H. Bhogan, Volley: Automated data placement for geo-distributed cloud services, in: Proc. of the 7th USENIX Conf. on Networked Systems Design and Implementation, in: NSDI'10, 2010.
- [5] F. Bonomi, R. Milito, P. Natarajan, J. Zhu, Fog computing: A platform for internet of things and analytics, in: N. Bessis, C. Dobre (Eds.), Big Data and Internet of Things: A Roadmap for Smart Environments, 2014.
- [6] K. Simon, Project natick-microsoft's self-sufficient underwater datacenters, 2018.
- [7] P. Mach, Z. Becvar, Mobile edge computing: A survey on architecture and computation offloading, IEEE Commun. Surv. Tutor. 19 (3) (2017).
- [8] J.C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J.J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, D. Woodford, Spanner: Google's globally distributed database, ACM Trans. Comput. Syst. 31 (3) (2013).
- [9] W. Lloyd, M.J. Freedman, M. Kaminsky, D.G. Andersen, Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS, in: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, in: SOSP '11, Association for Computing Machinery, New York, NY, USA, 2011, pp. 401–416.

- [10] K.P. Birman, The process group approach to reliable distributed computing, *Commun. ACM* 36 (12) (1993).
- [11] S. Jamin, C. Jin, A.R. Kurc, D. Raz, Y. Shavitt, Constrained mirror placement on the internet, in: *Proc. IEEE INFOCOM 2001. Conf. on Computer Communications. 20th Annual Joint Conf. of the IEEE Computer and Communications Society*, Vol. 1, Cat. No.01CH37213, 2001.
- [12] M. Karlsson, M. Mahalingam, Do we need replica placement algorithms in content delivery networks, in: *Proc. of the Int'L Workshop on Web Content Caching and Distribution, WCW, 2002*.
- [13] L. Gruenwald, M.H. Eich, Choosing the best storage technique for a main memory database system, in: *Procs. of the 5th Jerusalem Conf. on Next Decade in Information Technology*, 1990.
- [14] C. Curino, E. Jones, Y. Zhang, S. Madden, Schism: A workload-driven approach to database replication and partitioning, *Proc. VLDB Endow.* 3 (1–2) (2010).
- [15] G. Karypis, V. Kumar, Multilevel-way partitioning scheme for irregular graphs, *J. Parallel Distrib. Comput.* 48 (1) (1998).
- [16] R. Halalai, P. Sutra, É. Rivière, P. Felber, ZooFence: Principled service partitioning and application to the ZooKeeper coordination service, in: *2014 IEEE 33rd Int'L Symp. on Reliable Distributed Systems*, 2014.
- [17] P. Hunt, M. Konar, F.P. Junqueira, B. Reed, ZooKeeper: Wait-free coordination for internet-scale systems, in: *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, in: *USENIXATC'10*, USENIX Association, USA, 2010, p. 11.
- [18] O. Asad, B. Kemme, AdaptCache: Adaptive data partitioning and migration for distributed object caches, in: *Procs. of the 17th Int'L Middleware Conf.*, in: *Middleware '16*, 2016.
- [19] S. Elnikety, S. Dropsho, W. Zwaenepoel, Tashkent+: Memory-aware load balancing and update filtering in replicated databases, in: *Procs. of the 2nd ACM SIGOPS/EuroSys European Conf. on Computer Systems 2007*, in: *EuroSys '07*, 2007.
- [20] V.S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, E. Nahum, Locality-aware request distribution in cluster-based network servers, in: *Procs. of the 8th Int'L Conf. on Architectural Support for Programming Languages and Operating Systems*, in: *ASPLOS VIII*, 1998.
- [21] A. Pavlo, C. Curino, S. Zdonik, Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems, in: *Procs. of the 2012 ACM SIGMOD Int'L Conf. on Management of Data*, in: *SIGMOD '12*, 2012.
- [22] M. Jelasity, A. Montresor, O. Babaoglu, T-man: Gossip-based fast overlay topology construction, *Comput. Netw.* 53 (13) (2009) 2321–2339.
- [23] J. Leitão, J.O. Pereira, L. Rodrigues, Epidemic broadcast trees, in: J. Huai, R. Baldoni, I. Yen (Eds.), *26th IEEE Symposium on Reliable Distributed Systems - SRDS*, IEEE Computer Society, Beijing, China, 2007, pp. 301–310.
- [24] R. Ariyatt, F. Taiani, Fluidify: Decentralized overlay deployment in a multi-cloud world, in: *DAIS, INRIA Grenoble, France*, 2015, p. 14.
- [25] M.S. Ardekani, D.B. Terry, A self-configurable geo-replicated cloud storage system, in: *Proc. of the 11th USENIX Conf. on Operating Systems Design and Implementation*, in: *OSDI'14*, 2014.
- [26] L.L. Hoang, C.E. Bezerra, F. Pedone, Dynamic scalable state machine replication, in: *46th IEEE/IFIP Int'L Conf. on Dependable Systems and Networks, DSN, 2016*.
- [27] D. Sciascia, F. Pedone, Geo-replicated storage with scalable deferred update replication, in: *Procs. of the 2014 IEEE 33rd Int'L Symp. on Reliable Distributed Systems Workshops*, in: *SRDSW '14*, 2014.
- [28] R.A.C. da Silva, N.L. S. da Fonseca, On the location of fog nodes in fog-cloud infrastructures, *Sensors* 19 (11) (2019).
- [29] S. Mondal, G. Das, E. Wong, CCOMPASSION: A hybrid cloudlet placement framework over passive optical access networks, in: *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 2018, pp. 216–224, <http://dx.doi.org/10.1109/INFOCOM.2018.8485846>.
- [30] N. Cruz, H. Miranda, Recurring contacts between groups of devices: Analysis and application, *IEEE Trans. Mob. Comput.* 17 (7) (2018) 1666–1679.
- [31] H. Gauttam, K. Pattanaik, S. Bhaduria, D. Saxena, Sapna, A cost aware topology formation scheme for latency sensitive applications in edge infrastructure-as-a-service paradigm, *J. Netw. Comput. Appl.* 199 (2022) 103303.
- [32] M.T. Gonzalez-Aparicio, M. Younas, J. Tuya, R. Casado, Evaluation of ACE properties of traditional SQL and NoSQL big data systems, in: *Proc. of the 34th ACM/SIGAPP Symp. on Applied Computing*, in: *SAC '19*, 2019.
- [33] M. Satyanarayanan, The emergence of edge computing, *Computer* 50 (1) (2017) 30–39.
- [34] F.B. Schneider, Implementing fault-tolerant services using the state machine approach: A tutorial, *ACM Comput. Surv.* 22 (4) (1990).
- [35] L. Lamport, The part-time parliament, *ACM Trans. Comput. Syst.* 16 (2) (1998).
- [36] D. Ongaro, J. Ousterhout, In search of an understandable consensus algorithm, in: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, in: *USENIX ATC'14*, USENIX Association, USA, 2014, pp. 305–320.
- [37] F. Nawab, D. Agrawal, A. El Abbadi, DPaxos: Managing data closer to users for low-latency and mobile applications, in: *Proceedings of the 2018 International Conference on Management of Data*, in: *SIGMOD '18*, Association for Computing Machinery, New York, NY, USA, 2018, pp. 1221–1236.
- [38] C. Cachin, R. Guerraoui, L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*, second ed., 2011.
- [39] R. Friedman, R. van Renesse, Strong and weak virtual synchrony in horus, in: *Proc. of the 15th Symp. on Reliable Distributed Systems*, in: *SRDS '96*, 1996.
- [40] X. Défago, A. Schiper, P. Urbán, Total order broadcast and multicast algorithms: Taxonomy and survey, *ACM Comput. Surv.* 36 (4) (2004) 372–421.
- [41] K. Jenkins, K. Hopkinson, K. Birman, A gossip protocol for subgroup multicast, in: *Proc. 21st Int'L Conf. on Distributed Computing Systems Workshops*, 2001.
- [42] J. Sussman, I. Keidar, K. Marzullo, Optimistic virtual synchrony, in: *Proc. 19th IEEE Symp. on Reliable Distributed Systems SRDS-2000*, 2000.
- [43] Y. Amir, C. Danilov, J. Stanton, A low latency, loss tolerant architecture and protocol for wide area group communication, in: *Proc. Int'L Conf. on Dependable Systems and Networks. DSN 2000*, 2000.
- [44] H. Miranda, A. Pinto, L. Rodrigues, Appia, a flexible protocol kernel supporting multiple coordinated channels, in: *Proc. of the 21st Int'L Conf. on Distributed Computing Systems, ICDCS-21*, 2001.
- [45] M. Shapiro, N. Preguiça, C. Baquero, M. Zawirski, Conflict-free replicated data types, in: *Proc. of the 13th Int'L Conf. on Stabilization, Safety, and Security of Distributed Systems*, in: *SSS'11*, 2011.
- [46] C. Meiklejohn, H. Miller, Z. Lakhani, Towards a solution to the red wedding problem, in: *USENIX Workshop on Hot Topics in Edge Computing, HotEdge 18*, USENIX Association, Boston, MA, 2018.
- [47] M. Kleppmann, V.B.F. Gomes, D.P. Mulligan, A.R. Beresford, Interleaving anomalies in collaborative text editors, in: *Proc. of the 6th Workshop on Principles and Practice of Consistency for Distributed Data*, in: *PaPoC '19*, 2019.
- [48] P. Hunt, M. Konar, F.P. Junqueira, B. Reed, ZooKeeper: Wait-free coordination for internet-scale systems, in: *Proc. of the 2010 USENIX Conf. on USENIX Annual Technical Conf.*, in: *USENIXATC'10*, 2010.
- [49] B. Reed, F.P. Junqueira, A simple totally ordered broadcast protocol, in: *Proc. of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, in: *LADIS '08*, 2008.
- [50] S.H. Mortazavi, B. Balasubramanian, E. de Lara, S.P. Narayanan, Toward session consistency for the edge, in: *USENIX Workshop on Hot Topics in Edge Computing, HotEdge 18*, USENIX Association, Boston, MA, 2018.
- [51] H. Gupta, U. Ramachandran, FogStore: A geo-distributed key-value store guaranteeing low latency for strongly consistent access, in: *Proceedings of the 12th ACM International Conference on Distributed and Event-Based Systems*, in: *DEBS '18*, Association for Computing Machinery, New York, NY, USA, 2018, pp. 148–159.
- [52] D. Lima, H. Miranda, State deployment in fog computing, in: *MobiQuitous 2020 - 17th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, in: *MobiQuitous '20*, Association for Computing Machinery, New York, NY, USA, 2020, pp. 38–47.
- [53] B.B. Glade, K.P. Birman, R.C.B. Cooper, R. van Renesse, Light-weight process groups in the Isis system, *Distrib. Syst. Eng.* 1 (1) (1993).
- [54] K.P. Birman, Replication and fault-tolerance in the ISIS system, in: *Proc. of the Tenth ACM Symp. on Operating Systems Principles*, in: *SOSP '85*, 1985.



Diogo Lima is a final year Ph.D. student at the Faculty of Science of the University of Lisbon, Portugal. He is integrated in the LASIGE research unity and he is currently a teaching assistant at the Higher School of Tourism as well. His master's degree was on broadcasting algorithms for ad-hoc networks and then, in his Ph.D., his research focused on distributed systems, data deployment and consistency in Fog Computing. Throughout his Ph.D. he have published the following papers:

- Partial Replication Policies for Dynamic Distributed Transactional Memory in Edge Clouds, MECC 2016
- Simulation of partial replication in Distributed Transactional Memory, 2017 Wireless Days
- Weighting Past on the Geo-Aware State Deployment Problem, WoWMoM 2018
- Support of strong consistency on fog applications, PaPoC 2019
- State deployment in fog computing, MobiQuitous'20
- Applicability of Lightweight Groups to Fog Computing Systems, WiMob 2021