Applicability of Lightweight Groups to Fog Computing Systems

Diogo Lima[1,2]
[1] Escola Superior de Hotelaria
e Turismo do Estoril, Portugal
Email: dlima@lasige.di.fc.ul.pt

Hugo Miranda[2]
[2] LASIGE, Faculdade de Ciências,
Universidade de Lisboa, Portugal
Email: hamiranda@ciencias.ulisboa.pt

***Abstract*—The tradeoff between scalability and consistency has been addressed by deploying distributed applications in the cloud. However, cloud-based 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 computing power on surrogate servers at the network edge. To improve performance, one must deploy each fragment of the application state at the most convenient surrogate and maintain an efficient coordination when manipulating state stored in multiple locations. This paper formalizes a two-tier view synchronous group communication model for Fog Computing to improve performance of large scale mobile distributed applications and evaluates it using data derived from a real dataset.**

***Index Terms*—Mobile Distributed Applications, Fog Computing, Middleware, Consistency, Lightweight Groups**

I. Introduction

Large scale mobile distributed applications such as massive online augmented reality games (e.g. Ingress), tourism applications (e.g Yelp), or traffic management in smart cities [1] are required to manage a large number of shared objects (the application state), while providing a consistent view to every participant. The current trend is to centralize the application state in large scale datacenters in the Cloud, using frameworks such as Google Firebase.[1] This simplifies state management, but performance suffers from the geographical barrier between the end users and the application state, due to network latency and jitter.

Fog Computing [2] addresses this problem by approximating the cloud to the users with the deployment of *surrogate servers*. Surrogates provide the computing power and storage space required to manage application state at the network edge. Best performance can be achieved if the application state is divided by surrogates according to its usage and location patterns. However, state distribution by surrogates raises a number of challenges that could not be found in the centralized cloud approach. One is location management: deciding the most suitable location for state components, coordinate migrations, and facilitate their location by other participants (mobility). The second is concerned with keeping performance at an acceptable level considering the increasing number of participants in distributed transactions (distributed environment).

[1]https://firebase.google.com/

A promising solution relies on organizing the surrogates into groups arranged according to the state they store and manipulate. Virtual synchrony [3] is a good candidate as it organizes processes into groups and guarantees that messages are delivered to either all correct group members or to none at all, simplifying the satisfaction of any consistency requirements. However, virtual synchrony suffers from scalability problems and was not originally designed considering state mobility. This paper explores how virtual synchrony can be tailored to Fog Computing by extending the original concept with dynamically defined fine grained virtual synchronous groups. This technique, named LightWeight Groups (LWG) [4], [5], has the advantage of being adaptable to user mobility, as member affiliation costs are reduced to a small fraction of those in the original *heavyweight* group. The paper's main contribution is the formalization and implementation of such a two-tier view synchronous group communication abstraction for Fog Computing. Evaluation uses data derived from a real dataset to demonstrate the adaptability of the LWG solution to different application usages patterns and its scalability in the Fog Computing environment.

II. System Model and Problem Statement

Throughout this paper we assume a scenario exemplified by a large-scale forest surveillance system where heterogeneous devices (ground sensors, UAVs, etc.) cooperate on early detection of wild fires, hikers search and rescue or illegal logging. Devices communicate with Ground Stations (GS), strategically deployed over the surveyed site, using any available network technology including delay-tolerant network (DTN) to circumvent network coverage limitations. The GS represent the surrogates servers of the Fog Computing model, acting as entry points of the surveillance devices to the system. But besides collecting the large amount of data the latter produce, GS are also required to make split second decisions, e.g. to divert UAVs from their predefined route to investigate a smoke alarm or coordinate UAVs to avoid mid-air collisions.

Implementing a system with these characteristics is challenging. The real time constraints that are required for communication suggest that the most favorable approach is to transfer for each GS the control of the devices nearby as well as the management of the large amounts of data collected. However, decisions must be coordinated with nearby GS,

978-1-6654-2854-5/21/$31.00 ©2021 IEEE

for example if the rerouting of UAVs has the potential to create mid-air collisions or is unclear to which region can be attributed a smoke alarm. Given that all these scenarios depend on the state collected by the devices, the model further assumes that state can be partitioned in 3 categories. *Global* state is the information that is relevant to all participants in the system (GS, UAVs, command center), such as the current warnings/alarms and their possible cause. *Geo-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. In our example, good examples of Geo-dependent state are the set of measurements registered by the UAVs (i.e. humidity levels, temperature, wind), as well as the current number and location of UAVs and sensors in that area. Finally, *mobile* state refers to the information specific to the mobile devices such as current battery level or flight autonomy.

The approaches to target each of this data items to one category and location is discussed elsewhere (e.g. [6]) and outside the scope of this paper. Instead, this paper focus on the challenge of keeping the replicas of these items consistent in spite of multiple, possibly concurrent, updates provided by distinct GS. This paper addresses this problem by revisiting the virtual synchrony abstraction.

III. Related Work

Data consistency management can be centralized or decentralized. In contrast with some Fog Computing approaches that delegate consistency management on centralized Cloud-based SQL and NoSQL big data systems [7], this section discusses decentralized approaches. These are believed to cope better with the split second decisions required by applications like the surveillance application example.

A. Group View Synchronous Communication

The State Machine Replication (SMR) abstraction [8] applies 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 operations are delivered in the same order to every replica in a faulty environment, a concept coined as Atomic Broadcast. It has been shown that atomic broadcast requires a consensus algorithm, such as Paxos [9] or Raft [10]. Unfortunately, consensus algorithms are well-known for their scalability problems and addressing the scalability limitations of consensus necessarily implies a compromise elsewhere.

DS-SMR [11] 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, this model 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.

DPaxos [12] is a variant of Paxos intended to be used as the SMR component for Fog Computing systems and addresses scalability by proposing smaller and dynamic leader election quorums. It considers a geographically setting divided into zones and proposes *Zone-centric quorums* instead of 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, both DS-SMR and DPaxos exploit access locality and are optimized for workloads where operations do not frequently access objects in multiple partitions.

Virtual synchrony [3] simplifies consensus implementation by restricting the fault model and organizing processes into groups. 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 [13] 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 and, within that group view, the message must be delivered to all correct group view members, or to none at all. 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 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. There have been different frameworks to support virtual synchrony (e.g. [14], [15]), with different impacts on performance. Spread [14] relies on daemons to create virtual synchronous groups that aggregate all communication services. If a daemon crashes, all processes attached to it are affected. Appia [15], on the other hand, follows a decentralized approach where a crash only disconnects that process from the virtual synchony service.

B. Coordination in Internet-scale Systems

Apache ZooKeeper [16] is currently one of the most popular services for coordinating processes in distributed applications. To provide high availability, ZooKeeper replicates data on every server that composes the service. Read requests are executed locally by the server to which the client is connected, while write requests are forwarded to the leader instance and propagated from the leader to the other replicas through the Zab [17] atomic broadcast protocol. ZooKeeper is a good candidate for implementing our consistent multi group Fog Computing system model. However, scalability problems may arise if one surrogate is elected leader of multiple groups. Moreover, since each group would be coordinated by a different ZooKeeper instance, it is possible that one process may already been seen as faulty in some groups while still being considered correct in others, creating challenging consistency problems. In contrast, our approach guarantees that, as long as none of the processes crash, atomic broadcast is sufficient to perform write operations. In addition, our approach provides a consistent view of non-faulty processes.

Instead of a group approach, PathStore [18] considers a session consistency model between the mobile client and the closest replica where consistency is maintained in all

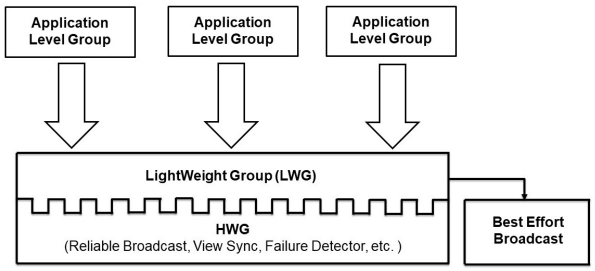


Figure 1: Two-tier group membership model.

operations performed within a session. Whenever the client changes location and reaches another replica, the latter contacts the initial replica to perform state synchronization. New operations are halted until this reconciliation procedure is finished. In opposition to this reactive approach, our solution relies in proactively creating groups of processes avoiding costly reconciliation procedures during client execution of an operation. The notion of groups of replicas is also exploited in FogStore [19], a key-value data store designed for low-latency accesses in 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 its *Context-of-Interest* (CoI) area. Consistency is maintained within a CoI and 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.

IV. Lightweight Group View-Synchronous Communication

This section presents a framework to provide virtual synchrony to Fog Computing applications by using the LightWeight Group (LWG) [4] abstraction. LWG extends the poorly scalable group view synchronous communication paradigm (HWG) with fine grained, application defined, subgroups while keeping performance at acceptable levels without compromising the properties provided by HWG. As depicted in Fig. 1, LWG rests on top of two building blocks. A Best Effort Broadcast (BEB) implementation will provide the fast (although unreliable) dissemination of each message. The HWG enforces the reliable delivery of messages and supports membership management. Along this section we refer the LightWeight Group communication service as LWG and $S_m = \{l_1, l_2, l_3, \dots\}$ is the set of lightweight group instances some member m is affiliated. The notation l_i refers to a single group instance from the previous set.

A. LWG API

Applications interact with the LWG service using the interface depicted in Table I. The *l* parameter associates each operation to one of the existing groups in S_m. The API presents two categories of operations: group membership management and message exchange. The latter is composed of the *Send* and *Receive* operations, which include a message *m* to be broadcast to the lightweight group *l*.

Operation	Description
Join(*l*)	Request to join group with name *l*.
Leave(*l*)	Request to leave group with name *l*.
Block(*l*)	Indicates the app to stop broadcast new messages in group *l* until a View or Unblock is received.
BlockOk(*l*)	Confirmation that no message will be broadcast.
View(*l*)	Indicates the app of a new view installed for *l*.
Unblock(*l*)	Indication that new message broadcast can resume.
Send(*m, l*)	Sends a message *m* within group *l*.
Receive(*m, p, l*)	Delivers a message *m* broadcast by process *p* in *l*.

Table I: Service API of the LWG abstraction

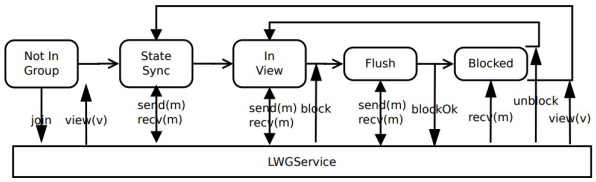


Figure 2: Evolution of application states when interacting with the LWG service.

Messages can only be exchanged once a process is a member of a group *l* and a view is installed. Fig. 2 depicts the application states when interacting with the LWG communication service. The application starts by expressing its intention to become member of a group with the *Join* operation. Eventually, a *View* indication is received to confirm the group membership. At this point, the application can broadcast messages, with the service guaranteeing the reliable or atomic broadcast properties. The delivery of a *View* indication further signals the application that a synchronization of the group state (*State Sync*) may be advisable. 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 some ongoing operations. Once the application acknowledges this indication with the *BlockOk* request, it enters the *blocked* state where it can still receive the messages sent from the *flush* procedure of other members but it is not allowed to reply to them. Transition from the *blocked* state happens 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 that led to the installation of a new view or the membership remained the same. Avoiding unnecessary view changes and group state synchronizations as provided by *Unblock* is key for the resource and performance gains of our approach.

B. Implementation

The lightweight group service is implemented with an instance of each of the LWG, HWG and BEB building blocks (depicted in Fig. 1). 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 the LWG instances of each

surrogate. BEB is used to provide fast application message delivery delaying the verification of the reliable broadcast properties until a view change occurs.

1) View Management: The service replicates the views state (defined by an ordered list of members and view id) of every l_i by all members of the heavyweight group. Consistent replication is guaranteed by using the HWG reliable broadcast service to disseminate all the *Join* and *Leave* requests and to coordinate the installation of lightweight group views. 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. As soon as the new heavyweight group view is installed, the 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 reliable broadcast service of the HWG to disseminate the leader decisions. This algorithm is an extension of the *flush* procedure defined in [20] 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 performance 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 addressed before when the view blocks and then unblocks as no change in the lightweight group membership was found.

A particular case happens when the leader of the heavyweight group fails. This problem is solved considering that: i) the HWG service elects a new leader with every view; and ii) the reliable broadcast service ensures that the view state of every lightweight group is consistent and can be synchronized with joining processes. As a result, the new leader can perform the coordination tasks as soon as the new heavyweight group view is installed. The period between the reception of the Block indication from the HWG service and the installation of new views or the unblock of the existing ones at the lightweight groups is mostly spent ensuring the delivery guarantees of the messages exchanged in the view.

2) Message broadcasting and consistency: An application data message m sent for group l_i is 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. The advantage of using the BEB service is that it scales linearly with the number of group members. However, it does not ensure the reliable broadcast properties, that must be found in any virtual synchrony service. 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_{S,l_i}.

A lightweight or heavyweight group is considered blocked when processes are not allowed to send messages. Once the group is blocked, proceedings to ensure reliable broadcast properties are as follows. At the lightweight group level, the LWG service on surrogate S uses the reliable broadcast primitive of the HWG to disseminate the content of its M_{S,l_i} set. The properties of reliable broadcast ensure that the union of the sets M_{S,l_i} received from every correct member of the lightweight group contains all messages delivered to any correct member of the view. LWG services can then compare their lists of messages with the union and deliver any message not delivered by BEB.

V. Evaluation

To assess the performance gains of the LWG service over virtual synchrony in Fog Computing, both were compared in two dimensions: scalability and performance. The goal is to understand if LWG can attenuate the well known scalability and performance limitations attributed to HWG. The implementation rests on top of the off-the-shelf View-Synchronous Group Communication service of *Appia* [15]. The tests using *LWG* differ from the *HWG* with the inclusion in the *Appia*'s composition stack of the *lwg* layer.

A. Testbed Configuration

To emulate the Fog based mobile distributed application, two transaction workloads were generated from the Austin's b-cycle (https://austin.bcycle.com/) application public dataset. Austin b-cycle is a bicycle sharing service where users checkout and return bicycles from specific stations (called *kiosks*). The baseline testing workload, hereafter named *Small Groups* (*SG*), consists on 90% of transactions that either checkout or return a bicycle to a kiosk, 5% to checkout a bicycle from a kiosk and reserve a parking slot at the destination and the remaining 5% to probe kiosks on a 500m radius for an available bicycle and make the respective checkout. A second workload was derived, hereafter named *Large Groups* (*LG*), composed of the opposing ratio of transaction distribution: 90% of transactions having at least three objects and the remaining 10% requiring at most two. Each transaction is implemented as a single message, delivered to the closest surrogate, who is in charge of committing the transaction. Considering the smaller number of participants per transaction, it is expected that the *SG* workload further benefits from the *lwg* approach.

Experiments were run with 6 to 24 surrogates, each participating on the fog computing service as an HWG member. Each object is stored and replicated in a lwg of 3 surrogates and the locations of each application state item is known in advance. Each surrogate runs on a *t2.micro* Amazon EC2 instance. Latency is simulated by distributing the instances by 5 Amazon datacenters in Europe. Members of the same replication group

are co-located in the same datacenter, to simulate the low latency connection between nearby surrogates.

The evaluation consists on comparing two approaches. In the *lwg* approach, surrogates are all part of the large virtual synchronous group on top of which fine grained lightweight groups are created by the LWG service. Lightweight groups are created when needed and between all members that store at least one object in the respective transaction. In the *hwg* approach, each transaction is exchanged in the single core group and delivered to every member. On average, each transaction carries 40 bytes of raw, non-compressed data. A test consists on having each surrogate delivering 150 000 transactions from either the *SG* or *LG* datasets. For *hwg*, the performance results are the same for both datasets as the number of surrogates involved in each transaction is constant. Performance is compared by measuring the time from the moment the test starts until the last surrogate receives the last message. Tests were executed in sequence in an effort to attenuate latency variations with the time of day. Results presented are the average 90th percentile of execution time of 20 runs in comparable conditions.

B. Crash Safe Steady Broadcast Test

In this test each surrogate broadcast a new transaction every $2ms$ and surrogates do not fail. Expectations are that broadcasting messages to different subgroups in parallel emphasises the *lwg*'s scalability in contrast with the *hwg* approach where all members are required to process every message. The results depicted in Fig. 3 show that the workload has a non-negligible impact on performance. The LWG abstraction can be up to 3 times faster than the HWG approach in the SG scenario. For the LG scenario, which increases group sizes due to larger multi-object transactions, *lwg* is still able to reduce execution time up to 60%. This behavior is explained by the fact that every additional member that joins the *hwg* configuration represents a fixed increase of 150000 messages while, in the *lwg* approach, system size increase leads to a more widespread data deployment. In turn, this creates a greater number of groups that can process transactions in parallel, leading to the *lwg*'s linear growth in both workloads. Nevertheless, an interesting observation from the figure is that the *hwg*'s completion times only start to differentiate from *lwg* at 9 surrogates. Until that point, all configurations present results very close to $300s$. This value is the minimum for test completion time considering that 150000 transactions are broadcast by each surrogate at a pace of one every $2ms$. This indicates that for smaller system sizes and until a message workload delivery of around 1 million per surrogate, the tests are able to finish as soon as the last message is broadcast.

C. Crash-Recovery Steady Broadcast Test

The behavior of *lwg* in a faulty environment was evaluated in scenarios where at every $60s$ each surrogate independently decides to crash and immediately recover with a probability of respectively 10% and 30%. After each crash, state consistency is recovered using the same synchronization procedure on the

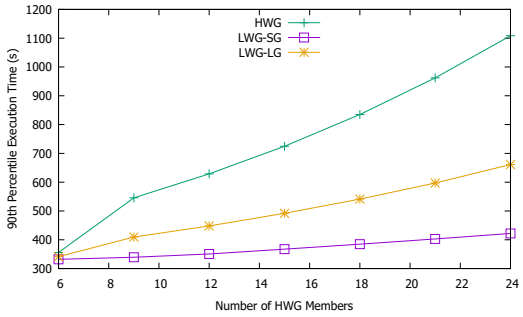
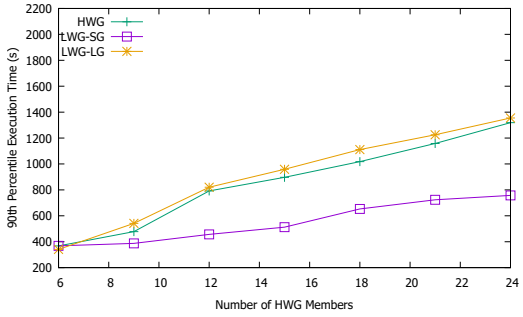
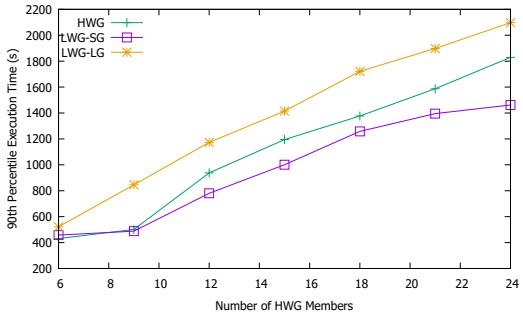


Figure 3: Completion time in the Crash Safe Steady Test.



(a) 10% probability of crashing



(b) 30% probability of crashing

Figure 4: Completion time in the Crash-Recovery Steady Test.

3 tests. Transactions are resumed once consistency is assured. Results are depicted in Fig. 4 and, as expected, show that faults have a non-negligible impact on the performance of all approaches. As discussed before, LWG concentrates most of its effort towards reliability on view installation time (a process that occurs once for every fault) and has a strong dependency on the underlying heavyweight group. Therefore, it would be expected that, at some point, the overhead introduced by the lwg approach would penalize its performance up to the point where the plain use of hwg would show to be more beneficial. However, this expectation is only half confirmed. Considering the 10% crash scenario, one can observe that *lwg-SG* takes about half the time to conclude the tests compared to the other two approaches. This is attributed to the fact that each crash only affects groups with smaller memberships, able to deliver new views faster and with a reduced number of undelivered messages that need to be rebroadcast before the test resumes.

This behavior proves that if the 2 conditions of Fog Computing are met: a well suited state deployment that promotes data locality and a small number of participants per transaction, the performance can be significantly improved over traditional view-synchronous systems, even in the presence of crashes. Since these later conditions are progressively lost in the *lwg-LG*, its performance is penalized.

D. 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 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 shows that, by creating multiple subgroups with smaller affiliations than the *hwg* coarse group, the *lwg* approach always presents better completion times regardless of the characteristics of the workload. But such solution is only possible for applications that do not require every transaction to be delivered to every member. In the presence of frequent faults, the performance gains obtained by the *lwg* approach were expected to be overtaken by the *hwg* counterpart, as it relies on the properties of the later to provide its own failure recovery. However, the results have shown that this expectation holds just to a certain degree. Even in the presence of a high degree of crashes (surrogates failing independently after every 300 000 transactions on average) and in face of a workload that exploits the validation of transactions by smaller groups, *lwg* is still able to outperform *hwg*. As any of the conditions that make *lwg* advantageous for Fog Computing degrade, the *lwg* gains are unavoidably erased, resorting to a performance comparable or worst to that of *hwg*.

VI. Conclusions

Large scale mobile distributed applications must manage the application state and provide a consistent view to every participant. Centralizing the application state in the Cloud facilitates these tasks, but creates a geographical barrier between the end users and the datacenters. Fog Computing tries to mitigate this impact by deploying surrogate servers at the network edge. However, its performance depends of the ability to deploy each of the application state components at their most convenient location and of a coordination mechanism to cope with distributed transactions. In this paper, we explore how virtual synchrony can benefit of partial groups defined at the application level to create multiple fine grained virtual synchronous groups, in a technique named Light-Weight Group (LWG). This paper formalized the LWG communication abstraction and presented an implementation. The evaluation has shown the benefits of the LWG abstraction in mitigating the scalability problems of the View-synchronous group communication (HWG). In a scenario without crashes, it is able to surpass the standard approach completion time by 2 to 3 times. In the presence of crashes, the LWG abstraction does not incur in a significant performance penalty if 2 conditions hold: percentage of crashes is not high and the LWG abstraction is able to fully exploit the existence of smaller groups.

Acknowledgment

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

References

[1] M. Gerla, "Vehicular cloud computing," in *2012 The 11th Annual Mediterranean Ad Hoc Networking Workshop (Med-Hoc-Net)*, 2012.
[2] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, *Fog Computing: A Platform for Internet of Things and Analytics*, 2014.
[3] K. P. Birman, "The process group approach to reliable distributed computing," *Commun. ACM*, vol. 36, no. 12, 1993.
[4] B. B. Glade, K. P. Birman, R. C. B. Cooper, and R. van Renesse, "Light-weight process groups in the isis system," *Distributed Systems Engineering*, vol. 1, no. 1, 1993.
[5] L. Rodrigues, K. Guo, P. Veríssimo, and K. Birman, "A dynamic light-weight group service," *Journal of Parallel and Distributed Computing*, vol. 60, no. 12, 2000.
[6] D. Lima and H. Miranda, "State deployment in fog computing," in *Proc. of the 17th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (to appear)*, ser. MobiQuitous '20, 2020.
[7] M. T. Gonzalez-Aparicio, M. Younas, J. Tuya, and 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*, ser. SAC '19, 2019.
[8] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, 1990.
[9] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, 1998.
[10] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. USENIX Association, 2014.
[11] L. L. Hoang, C. E. Bezerra, and F. Pedone, "Dynamic scalable state machine replication," in *46th IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN)*, 2016.
[12] F. Nawab, D. Agrawal, and A. El Abbadi, "Dpaxos: Managing data closer to users for low-latency and mobile applications," in *Proc. of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. Association for Computing Machinery, 2018.
[13] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*, 2nd ed., 2011.
[14] Y. Amir, C. Danilov, and 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.
[15] H. Miranda, A. Pinto, and 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.
[16] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *Proc. of the 2010 USENIX Conf. on USENIX Annual Technical Conf.*, ser. USENIXATC'10, 2010.
[17] B. Reed and F. P. Junqueira, "A simple totally ordered broadcast protocol," in *Proc. of the 2Nd Workshop on Large-Scale Distributed Systems and Middleware*, ser. LADIS '08, 2008.
[18] S. H. Mortazavi, B. Balasubramanian, E. de Lara, and S. P. Narayanan, "Toward session consistency for the edge," in *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*. USENIX Association, 2018.
[19] H. Gupta and U. Ramachandran, "Fogstore: A geo-distributed key-value store guaranteeing low latency for strongly consistent access," in *Proc. of the 12th ACM Int'l Conf. on Distributed and Event-Based Systems*, ser. DEBS '18, 2018.
[20] K. P. Birman, "Replication and fault-tolerance in the isis system," in *Proc. of the Tenth ACM Symp. on Operating Systems Principles*, ser. SOSP '85, 1985.