A Self-stabilizing Control Plane for Fog Ecosystems

Zacharias Georgiou*, Chryssis Georgiou*, George Pallis*, Elad M. Schiller†, Demetris Trihinas‡
* Department of Computer Science, University of Cyprus. Email: {zgeorg03, chryssis, gpallis}@cs.ucy.ac.cy
† Computer Science and Engineering, Chalmers University of Technology. Email: elad@chalmers.se
‡ Department of Computer Science, University of Nicosia. Email: trihinas.d@unic.ac.cy

Abstract—**Fog Computing is now emerging as the dominating paradigm bridging the compute and connectivity gap between sensing devices and latency-sensitive services. However, as fog deployments scale by accumulating numerous devices interconnected over highly dynamic and volatile network fabrics, the need for self-healing in the presence of failures is more evident. Using the prevailing methodology of self-stabilization, we propose a fault-tolerant framework for control planes that enables fog services to cope and recover from a very broad fault model. Specifically, our model considers network uncertainties, packet drops, node fail-stops and violations of the assumptions according to which the system was designed to operate (e.g., system state corruption). Our self-stabilizing algorithms guarantee automatic recovery within a constant number of communication rounds without the need for external (human) intervention. To showcase the framework's effectiveness, the correctness proof of the self-stabilizing algorithmic process is accompanied by a comprehensive evaluation featuring an open and reproducible testbed utilizing real-world data from the smart vehicle domain. Results show that our framework ensures a fog system recovers from faults in constant time, analytics are computed correctly, while the control plane overhead scales linearly towards the IoT load.**

Index Terms—**Fog Computing, Fault-Tolerance**

I. Introduction

Fog and Edge Computing are the technologies enabling computation at the network extremes, such as on downstream data, on behalf of cloud services, and upstream data, on behalf of IoT services [1]. The rationale of fog computing is that computing should happen at the proximity of the data source with the "fog" constituting any compute and network resources along the path between the data and the cloud. In this context, the "edge" differs from traditional sensing devices in that sensory data are processed in proximity and converted from raw signals to contextually relevant information [2]. In light of this, recent advancements in fog computing suggest using *cloudlets* as intermediate compute platforms between IoT (edge) devices and the cloud, enabling users to exploit the analytic power of the cloud without incurring high communication latency [3]. A cloudlet (also referred as a foglet, microcloud) can be a single server or a small cluster of co-located servers that form a (virtual) pool of shared resources but from an external viewpoint are considered a single entity [4]. Compared to traditional datacenters, a cloudlet features much more limited resources, albeit its proximity to IoT devices makes it appealing for offloading compute tasks and receiving timely responses.

Although fog computing brings the computation closer to delay-sensitive services, the challenges restricting the cloud paradigm still remain as the pace of generated data continues to rise [5]. Now, these overwhelming volumes of data not only have to be processed in time, but must be processed on, arguably, "weaker" hardware with potential nodes being wifi access points, drones, cameras, and even wearables. Also, fog infrastructure usually operates in geo-distributed and less controlled settings, with many applications competing for limited resources against high-priority services [6]. Consequently, failures due to hardware limitations and network uncertainties are highly likely at the fog continuum [7]. To maintain high availability, fog infrastructure must be resilient to both node and network failures. Thus, self-managing and self-healing solutions are required. IoT services must be able to recover from any issues that arise during their lifetime. In this context, it is critical to ensure continuous operation and recoverability at scale even in the event of failure without human involvement. In particular, cloudlets must satisfy the increasingly stringent fault-tolerance specifications of today's internet-enabled systems. In the current fog computing paradigm, fault-tolerance must be implemented to both preserve the system state locally at the edge and ensure the accuracy of analytics computations, especially in the case of a node failure or intermittent long-distance network connectivity problems.

We propose to address the challenge of dependable fog computing by using a fault-tolerant control plane that ensures service availability and data freshness in spite of the dynamic nature of the fog continuum. Via inter-connection of IoTs (edge devices), cloudlets and remote clouds, the proposed solution can tolerate network uncertainties, communication drops as well as cloudlet and IoT failures. In addition to these benign failures, our algorithms follow a very strong notion of fault-tolerance, called *self-stabilization* [8], which has provided the Internet with automatic failure recovery as early as the 1980's. Self-stabilization ensures that the fog can recover after the occurrence of any temporary violations to the assumptions according to which the system was designed to operate. These violations can include state corruption, extreme number of node failures, network partitions or unexpected system reconfiguration. Once such transient violations occur, non-self-stabilizing systems cannot guarantee correct system behaviour or that analytics queries respond correctly due to the loss of data or the propagation of corrupted information. In order for a system to be considered "self-stabilizing", a correctness proof is required to guarantee recovery, within finite time, after the occurrence of the last transient violation [9].

Contribution and Research Outcome. This paper addresses the problem of how to tolerate and recover from run-time faults in fog ecosystems. We consider a typical fog architecture, where edge devices are interconnected with remote clouds via network elements, denoted as cloudlets. Specifically:

- We introduce a self-stabilization framework integratable with distributed control planes for fog realms. The control plane is the core of the ecosystem and manages the network fabric with a global viewpoint and establishes the routing path of data serviced by geo-distributed cloudlets. By adopting our self-stabilization framework, the control plane can cope with an even broader fault model than the one that includes just communication and node failures.
- Our framework is accompanied by a comprehensive overview of the algorithmic mechanisms required by all involved system actors. Most importantly, our solution includes the correctness proof detailing system recovery, after transient faults, within a constant number of communication rounds. The current state-of-the-art in self-stabilization for IoT systems recovery time is bounded, at best, by the number of participating entities.
- To illustrate both the effectiveness and low runtime footprint of our framework at scale, we introduce a thorough evaluation using real-world data and actual queries of interest from an intelligent transportation service. Our results are reproducible and the reference implementation (including configuration and test data) is available on Github [1]. Our experiments validate our analysis and show that even in the presence of severe failures, our solution can always recover in constant time while the network overhead scales linearly towards the IoT load.

Paper organization. Section II reviews related research. Section III presents the system model and objectives. Section IV proposes the solution. Section V sketches the proof. The details appears in [10]. Section VI presents the experimentation, followed by the conclusion.

II. Related Work

Fog and edge infrastructures are typically composed by hundreds or thousands of heterogeneous and interacting components, leading to the emergence of different types of faults. A major challenge in fog computing is to define the fault and failure coverage for high QoS [11]. Faults may occur either simultaneously or in any aspect of system operations ranging from application to hardware, and may have several causes, including insufficient memory, performance interference, network congestion, server faults, application crashes, etc.

Due to these challenges, existing work on fault-tolerance in large-scale distributed systems often have limitations in terms of practicality and performance guarantees, as further documented. In [11], the authors introduce a framework providing consistency guarantees for stateful edge services by adopting a fault-tolerant middlebox using the classic approach of "rollback recovery" where a system uses information logged during normal operation to reconstruct state after failure. In [12], authors present a fault-tolerant messaging architecture for edge systems. This is achieved by introducing timing bounds that capture the relation between service parameters and loss-tolerance requirements. In [13], a fault-tolerant fog framework for data transmission is introduced. The proposed fault-tolerance mechanism combines the advantages of Directed Diffusion and Limited Flooding to enhance data transmission reliability. In [14], authors introduce a fault-tolerant algorithmic approach ensuring that collaborating fog nodes sharing compute resources do not become disjoint when a faulty node becomes unresponsive. This is achieved by establishing that each node maintains connectivity information of its neighbouring nodes with the key limitation being that supportive networks must adopt a flat topology model. None of these solutions provides a holistic approach for addressing fault-tolerance in the fog continuum.

Our framework fits naturally in control planes, *e.g.,* Istio and Linkerd [15], [16], that decouple operational control and policy enforcement from the business logic of distributed network fabrics. These frameworks provide fault-tolerance in the form of timeouts for labelling nodes as failed. In turn, circuit breaking is provided to safe-guard nodes overwhelmed by requests so that nodes "fail fast" when requests exceed a threshold. Thanks to our self-stabilizing algorithmic process, distributed control planes are introduced to a very strong notion of fault-tolerance on network uncertainties, communication drops, configuration errors, arbitrary transient violations, cloudlet and IoT fail-stop failures. In turn, no combination of faults can yield the system execution or corrupt data computations.

In the context of self-stabilization and IoT, Siegemund *et al.* [17] present a self-stabilizing pub/sub middleware for IoT services. Their basic idea is that fault-tolerance is ensured through the construction of a distributed self-stabilizing data structure based on a virtual ring. However, operations over this ring take $\mathcal{O}(n)$ time even in the absence of failures, where n is the ring size. Canini *et al.* [18] present a self-stabilizing control plane for software-defined networks (SDNs). Their work assumes that all nodes are either client hosts, switches or controllers. The algorithm stabilizes within $\mathcal{O}(d^2n)$, where d is the network diameter and n is the number of nodes. Chattopadhyay *et al.* [19] integrate an SDN control plane with the in-network processing infrastructure that can offload IoT services. They use a single centralized service deployment controller and lightweight SDN micro-controllers (μC). They mention that their algorithm for μC placement is self-stabilized with a linear convergence time but the provided proof does not consider the designed criteria of self-stabilization that was defined by Dijkstra [9] and clarified by Dolev [8]. We provide both analytical and empirical proof for convergence in constant time. The state-machine replication technique used in this paper is inspired by practically-self-stabilizing virtual synchrony [20]. Moreover, our self-stabilizing solution stabilizes in constant time whereas the one in [20] does not have a bounded stabilization time (by the definition of the solution criteria of practically-self-stabilizing systems).

While interesting and relevant, the above works do not address the impact of strong fault-tolerance in a hierarchical network organization that includes cloud infrastructure,

[1] https://github.com/UCY-LINC-LAB/Self-Stabilization-Edge-Simulator

cloudlets that are placed at the network edge and IoT devices. Our recovery time is within $\mathcal{O}(1)$ and our placement mechanism convergence is within $\mathcal{O}(1)$. We base our proofs on the definition of self-stabilizing systems [8]. The definition requires the system to use bounded memory and recover after the occurrence of any transient violation of the assumptions according to which the system was design to operate. To the best of our knowledge, we are the first to propose an $\mathcal{O}(1)$ self-stabilizing control plane for fog and edge ecosystems.

III. Problem and System Description

System and Actors. We consider a fog ecosystem comprised of sets of nodes, such as the one of cloudlets C and IoT devices S, as well as a remote cloud infrastructure, which we refer to as the Cloud. Each cloudlet features specified communication, computation, and storage capabilities. A cloudlet is associated with a wireless access point covering a local area, referred to as a cell. The cloudlets in C form a shared resource pool that can serve the system collaboratively, *e.g.,* aggregating IoT data and forwarding it to the Cloud. We assume that the cloudlets can share (over the Internet) such aggregated data with the Cloud by accessing a shared repository (e.g., message queue). The Cloud can use the repository to instruct cloudlets, *e.g.,* which queries to serve the IoTs (edge devices), or advise the cloudlets on how to organize themselves, *e.g.,* propose the most-suitable leader according to the cloudlet specified capabilities and statistics gathered by the Cloud. The cloudlets themselves are intra-connected by backhaul links. We assume that, in the absence of failures, the QoS of these links allow to send data and control messages in a timely manner. The control plane manages and configures the cloudlets to route traffic and enforce service placement with IoT devices.

Objectives. We aim at developing a fault-tolerant framework for control planes that enables fog ecosystems to cope with communication uncertainties and a broad fault model without downtime or the need for external (human) intervention. The latter is particularly important, as moving computations to the cloud, in case of frequent fog node failures, significantly impacts system reliability and the performance of the overall system due to the large communication latency.

- O1. The Cloud, cloudlets and IoT devices must exchange messages within a constant number of communication rounds (and using a constant number of messages) per information update.
- O2. The memory space and compute time of any system entity must always be bounded and network traffic scale linearly to the number of system entities.
- O3. The presence of a constant number of benign faults must not degrade the system performance beyond the bounds that are imposed by the system communication and processing delays. *I.e.,* objective O1 must not be violated in the presence of benign faults (and the absence of violations considered in objective O4).
- O4. We also consider arbitrary transient violations of the assumptions according to which the system was designed to operate (as long as the algorithm code stays intact). After the occurrence of these violations, the system must recover autonomously within a constant number of communication rounds and return to satisfy the task specifications. By autonomous we mean the absence of external (human) intervention.

Actor Specifications. The fog control plane organizes the *cloudlet layer* (Figure 1), such that the presence of communication and node failures cannot disrupt the execution of services, *e.g.,* IoT queries. In detail, we require the implementation of the following functionality:

The cloudlet and IoT *registration* allows the Cloud to include individual nodes in the system (Figure 1). A node is allowed, after a predefined delay and local cleanups, to register again when it notices that it became disconnected from the system due to failures. The latter case is rare, and thus, should not repeatedly consume system resources.

The *query* functionality allows the Cloud to request the flow of information according to a model that the IoTs (edge devices) are to update periodically. That is, given the Cloud's current belief about the query result, the specified IoTs will update the system whenever the collected sensory information deviates from the model [5]. The cloudlet aim here is to aggregate these updates so that a concise query result arrives to the Cloud. Since this needs to be done in the presence of communication and node failures, each IoT should send its updates to a set of cloudlets and the latter should acknowledge (Figure 2). The cloudlets then should use a *leader* to unify their updates and forward concise query results to the Cloud. The cloudlet layer must function well in case of a failing leader. Therefore, a set of cloudlets, called *guards*, should monitor the leader's activity and guarantee query result delivery until the system decides on a new leader (Figure 3).

The management of *general-purpose services* can help to overcome capability differences among individual nodes via task load-balancing. Such tasks can be initiated by IoT users that need to leverage cloudlet capabilities. Also, cloud services may wish to avoid network-intensive computations, such as virtual traffic light that base its decisions on the current road traffic conditions of different vehicles. The fault-tolerant management of such services can be based on state-machine replication that is well-synchronized with query operations.

IV. Proposed Solution

Algorithms 1, 2, 3 and 4 collectively propose the solution to the above task specifications by considering the protocol to be executed by the Cloud, IoT devices, cloudlets, and respectively, the emulators of the replicated state-machine. Algorithm 1 assumes the availability of a self-stabilizing cloud infrastructure, such as [21].

Overview. The Cloud periodically monitors the system and keeps track of the Cloudlets and running IoT devices. Based on this information, and according to some mapping, each cloudlet is associated with a list of IoT devices [4]. The IoT devices periodically send updates (e.g., sensory information) to their associated cloudlet(s). Instead of each cloudlet reporting directly to the cloud, cloudlets report collected data to a leader.

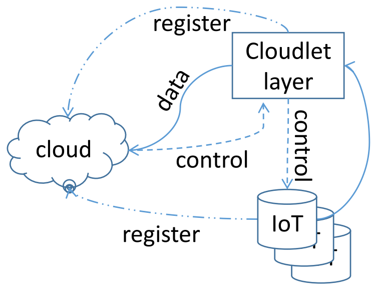

Fig. 1. System overview

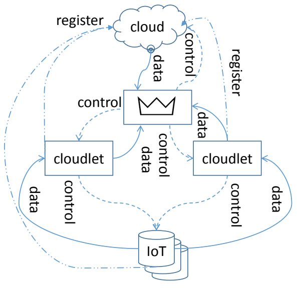

Fig. 2. The leader-based architecture

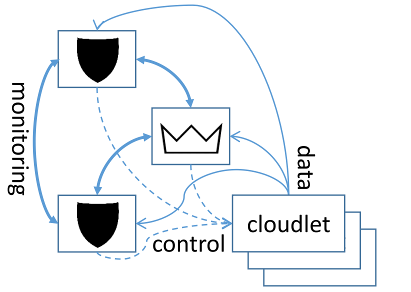

Fig. 3. Inner-structure of the cloudlet layer

The leader is the one that collects and aggregates all data and reports it to the Cloud. The above constitutes a "normal" (fault-free) operation. However, due to unexpected transient faults or more permanent faults (e.g., cloudlet fail-stop), as well as the need for bounded counters, additional checks must take place at different components of the system. Algorithms 1, 2, and 3 present such details for the Cloud, the IoT devices and the Cloudlets. Furthermore, in the event of a leader fail-stop, we do not want the data flow to the cloud to be suspended. To this respect, from the list of operational cloudlets, the Cloud also appoints a set of guards. The purpose of the guards is to monitor the status of the leader and in the event that the leader fail-stops, they report the latest collected data to the Cloud. Therefore, in Algorithm 3, each cloudlet reports not only to the leader, but also to the guards. Since the leader and the guards need to maintain consistent information on the collected data (and on any other information the control plane could be maintaining), they run Algorithm 4, which realizes a self-stabilizing state-machine replication mechanism.

In Section V we provide the correctness proof illustrating that our framework can self-stabilize in a constant number of communication rounds, while Section VI shows through a large testbed that there is no information loss even in the presence of multiple, different and randomly injected failures to the fog ecosystem. We now present more details, starting by describing the registers that are shared by nodes. Then, we go through the code according the above functionality list.

Registers. The shared register stores the aggregated sensory information collected by the IoTs (lines 24–25), aggregated by their cloudlets (line 56), and written by the leader (line 84). The Cloud and the cloudlet exchange control information via the shared registers $info$ and $infoAck$. The register $info$ includes the fields $devices$, $cloudlets$, $leader$ and $guards$. The register $infoAck$ is an array, such that the entry $infoAck[k]$ holds p_k's acknowledgment, where $p_k \in C$ is a cloudlet and the acknowledgment includes all the fields of $info$. In detail, the Cloud, $p_{cloudID}$, stores its view on the system membership in $info$ (line 15) and cloudlet p_k acknowledges the reception of this information by copying the value of $info$ to $infoAck[k]$ (line 47). Moreover, $p_{cloudID}$ selects, when needed, new cloudlets' leader (line 10) and guards (line 12).

Registration. IoTs and cloudlets register directly to the Cloud by sending a registration message (lines 24 and 48) after initializing their local variables and communication channels. This initialization guarantees that the joining node (or its communication channels) does not hold stale information. Once the registration message arrives to the Cloud, $p_{cloudID}$, the Cloud lists the joining node as a newcomer (lines 16 and 17). These newcomers will be listed as the system's IoT devices and cloudlets (lines 8–9) after the completion of the previous update round of these sets, which line 7 assures. The proposed solution assumes access to unreliable failure detectors. This allows the Cloud not to wait for cloudlets that are suspected to be faulty as well as to remove failing nodes from the IoT and cloudlet sets.

Query. We consider queries that are initiated by Cloud applications and require repeated updates. These queries include the Cloud current belief about the anticipated result, which we refer to as the query model. This allows IoT devices to reduce the number of times in which they transmit results to periodic queries since there is no need to transmit a result that fit the current belief of the Cloud according to the query model.

In detail, the registration procedure constructs up-to-date views on the sets of IoT $devices$ and $cloudlets$ in the shared register together with the current leader and guards. The proposed solution associates with each IoT the query description and model. This information is stored in $devices$. The cloudlets use a function, $myIoT()$, for mapping between them and the IoTs that they are responsible to communicate with (line 55). (A possible mapping could be to have the IoTs being assigned to the cloudlets in the same region, based on their proximity. Nevertheless, our system is independent on the specific mapping employed.) Cloudlets send the queries (along with their models) to these IoTs. The latter store the arriving information and acknowledge (lines 30–33). Once in a predefined periodicity, the IoTs update the query results, if needed (line 25). The cloudlets acknowledge the update arrival (lines 33 and 63). The cloudlets in turn periodically aggregate the sensory information received by the IoTs and send it to the leader and the guards (line 56). The leader updates the shared repository with the query results (line 84), whereas the guards serve as warm-backup leaders. We assume access to the functions $electLeader()$ and $selectGuards()$ that for a given set of system cloudlets elect a leader and select guards, respectively. In electing a leader and guards, we may want nodes that are more stealth, maybe closer to the IoT devices or in the center of the coverage area (*e.g.,* in the center of the city); the leader/guard selection problem can be inherent to the fog service placement problem (FSPP) [4], which is a different challenge in fog computing than the studied one. Nevertheless, in our system we could swap in/out FSPP algorithms and we are resilient to the algorithm in use.

State-machine replication. Since both the leader and the guards receive aggregated data from the cloudlets, they need to be in sync with respect to the data. More generally, the leader and the guards provide additional service as part of the control plane. So, they need to coordinate their activities and maintain consistent state between them. The fact that the system is asynchronous, together with the need for self-stabilization, makes it quite a challenging task. To this respect, we have the leader and the guards to run Algorithm 4.

The algorithm maintains a consistent state (aggregated sensory information) by performing multicast rounds coordinated by the leader. All necessary replica information (including the state) is maintained by each node in array $rep[]$ (line 72), which is exchanged between the leader and the guards (lines 102–104). In detail, once a cloudlet realizes that it has become the leader (line 89), it proposes to install a *view* of the current members, which includes itself and the guards that according to its local failure detector have not fail-stopped. The guards start following the leader towards installing this view by adopting its proposal (line 99). Once the leader sees that the view members have adopted its proposal (lines 92 and 74), it builds the new state based on the collected messages and states (lines 95 and 75) and proceeds to install the view. The guards adopt the leader's rep – including the (new) state (lines 98 and 79) completing in this way the installation of the view (lines 94 and 76). The multicast rounds can now begin, which are coordinated by the leader (lines 93 and 81–85) and followed by the guards (lines 97 and 80). The access to the application's message queue (commands to be executed by the state machine) is done via $fetch()$, which returns the next multicast message; the state transition function $apply(state, msg)$ applies the aggregated input array msg to the replica's $state$ and produces the local side effects. Simply put, in our case, the input to the state machine is the aggregated sensory information, which is sent by the cloudlets to the leader and the guards in Algorithm 3 (line 56) and stored by the latter in $agrregateinfo$ (line 66). So, essentially the multicast rounds of the state machine keep this information consistent among the leader and the guards. At the end of each multicast round, the leader updates the sensory information maintained in the shared register $data$ (line 84).

In the event of a leader fail-stop, and until the Cloud assigns a new leader (line 10), the guards update the shared registry (line 100). This ensures a continual update of the sensory information (which, depending on the application, could be crucial). If there is a change in the set of guards (either due to a fail-stop or due to an update of this set by the Cloud), then the leader begins the procedure to install a new view (line 90) with the new membership, without the need of any external intervention (including that of the Cloud). The failure detector abstraction (defined in line 72) can be implemented using heartbeats and counter thresholds (see for example [22]), or using "hello" messages and timeouts in a more time-informed setting (as we do in our simulation study in Section VI).

System state recovery via global reset. Self-stabilization requires bounded space, including bounded counters. Counters can grow up to a predefined size $MAXINT$, *e.g.,* $2^{64}-1$. Under normal operation, and if say, a counter is incremented every nano-second, then this limit can be reached in 146 years. However, a transient violation of the assumptions according to which the system was designed to operate can corrupt the counter. In this case (lines 27, 58, and 103), the cloudlet or IoT holding this counter will send a RESET message to the Cloud, calling for a global system reset. The Cloud, upon receiving such a message (line 18), initiates the *reset procedure*: it sets the shared register $info$ into $\bot$ (line 11 or line 18), and waits until all non-faulty cloudlets have acknowledged this, before it unregisters all cloudlets and IoT devices (by setting $info$ into $(\emptyset, \emptyset, \emptyset, \emptyset)$) and flashes all its local variables. This causes each cloudlet (line 48) to register again after a local reset of the node state and its communication channels, following the registration procedure described above. Since the IoTs are no longer in $info.devices$, no cloudlet will contact them, causing each (non-faulty) IoT to timeout and hence also register again after a similar initialization procedure (line 23).

V. Correctness Proof

Our analysis demonstrates a constant time recovery from arbitrary transient faults. It considers the *interleaving model* [8], in which the node's program is a sequence of *(atomic) steps*. Each step starts with an internal computation and finishes with a single communication operation, *i.e.,* message $send$ or $receive$. The *state*, s_i, of node $p_i \in \mathcal{P}$ includes all of p_i's variables and the set of all incoming communication channels. Note that p_i's step can change s_i as well as remove a message from $channel_{j,i}$ (upon message arrival) or add a message in $channel_{i,j}$ (when a message is sent). The term *system state* refers to a tuple of the form $c = (s_1, s_2, \cdots, s_n)$ (system configuration), where each s_i is p_i's state (including messages in transit to p_i). An *execution (or run)* $R = c_0, a_0, c_1, a_1, \ldots$ is an alternating sequence of system states c_x and steps a_x, such that each c_{x+1}, except c_0, is obtained from the preceding one, c_x, by the execution of step a_x. We say that execution R is legal if it satisfies the task specifications throughout R. We say that a system state c is safe if every execution that start from c is legal. Definition 5.1 considers a system state that Theorem 5.1 shows to be safe.

Definition 5.1 (Safe system state): We say that the system state c is safe if the following hold. (1) Let $p_i \in C$ and $p_j \in S$, such that $(j, m_j) \in myIoT(devices_{cloudID}, cloudlets_{cloudID})$. It holds that $cloudletList_j = cloudletList(j, cloudlets_{cloudID}) \land (lastUpdate_j \leq clock_j())$. Moreover, $devices_{cloudID} = \{(k, \bullet) \in deviceSet_i\} \land ((z, t, \bullet) \in \mathit{agreegateInfo}_j \implies p_z \in C \land t \leq clock_i()) \land (j, \bullet, m_j) \in \mathit{agreegateInfo}_j$. (2) The value of $msgseq_i$, $msgc_i$ and $msgtoiot_i$ is greater or equal to any value of $msgseq$, $msgc$, and respectively, $msgtoiot$ fields associated with p_i in messages and cloudlets.

Algorithm 1: Code for the self-stabilizing cloud $p_{cloudID}$.

```
1  Variables: $newCloudlet/newIot$: new cloudlets and IoTs and their models (bounded by $cloudletSetSize$); $sequence$: leadership number;
2  Shared registers: $data$: is a data structure that stores the sensory information, to be processed by the cloud depending on the application; it includes records of the form $(id, leader, round, dat)$, where $id$ is the cloudlet's unique id that included the context $dat$ in the data structure, at round $round$ of the state machine with leader $leader$; $info$: has the form of $(devices, cloudlets, leader, guards)$, where the field $devices$ is a set (bounded by $deviceSetSize$) of IoT devices, their models and the information needed for failure detection; $cloudlets$ is a set (bounded by $cloudletSetSize$) of cloudlets and the information needed for failure detection; $leader$ of the form $(seq, id)$ is the cloudlets' current leader and an associated sequence number; $guards$ is a set of cloudlets ids (a subset of $cloudlets$) that have been selected as guards; $infoAck[cloudletSetSize]$: an array that stores the latest value of $info$ that each cloudlet has read;
3  Interface: $suspectedIot(set)$ and $suspectedCloudlet(set)$: return the sets of suspected to be faulty IoT devices and cloudlets, respectively; $electLeader(set)$: returns the elected leader from $set$; $selectGuards(set)$: returns the set of guards from $set$;
4  do forever /* use predefined periodicity */ begin
5      let $lInfo := (lDevices, lCloudlets, lLeader, lGuards) :=$ read($info$);
6      let $lInfoAck :=$ read($infoAck$);
7      if $\bot \neq lInfo \wedge (\{lInfo\} = \{lInfoAck[k] : k \in C \setminus suspectedCloudlet(C)\})$ then
8          $(lDevices, newIot) \leftarrow ((lDevices \setminus \{(k, \bullet) : k \in suspectedIot(lDevices)\}) \cup newIot, \emptyset)$;
9          $(lCloudlets, newCloudlet) \leftarrow ((lCloudlets \setminus \{(k, \bullet) : k \in suspectedCloudlet(lCloudlets)\}) \cup newCloudlet, \emptyset)$;
10         if $lLeader.id \notin lCloudlets$ then
               $lLeader \leftarrow (sequence{+}{+}, electLeader(lCloudlets))$;
11         if $sequence = MAXINT$ then write($info, \bot$);
12         if $(lGuards \cap lCloudlets) = \emptyset$ then
               $lGuards \leftarrow selectGuards(lCloudlets \setminus \{lLeader.id\})$;
13         write($info, (lDevices, lCloudlets, lLeader, lGuards)$);
14     else if $\{\bot, (\emptyset, \emptyset, \emptyset, \emptyset)\} \supseteq (\{lInfo\} \cup \{lInfoAck[k] : k \in C \setminus suspectedCloudlet(C)\})$ then write($info, (\emptyset, \emptyset, \emptyset, \emptyset)$);
           $(newCloudlet, newIot, sequence) \leftarrow (\emptyset, \emptyset, 0)$;
15     else if $\{\bot\} \subset (\{lInfo\} \cup \{lInfoAck[k] : k \in C \setminus suspectedCloudlet(C)\})$ then write($info, \bot$);
16 upon message $m = \langle \mathsf{REGISTER} \rangle$ arrival from IoT $j$ at time $t$ do
       $newIot \leftarrow (newIot \cup \{(j, t, \bot)\})$;
17 upon message $m = \langle \mathsf{REGISTER} \rangle$ arrival from cloudlet $z$ at time $t$ do
       $newCloudlet \leftarrow (newCloudlet \cup \{(z, t)\})$;
18 upon message $m = \langle \mathsf{RESET} \rangle$ arrival from device $k$ do write($info, \bot$);
```

(3) $|A| = 1$, where $A = \{(v, su, r, sa, m) : p_i, p_j \in C \wedge (v, su, r, sa, m, \bullet) = r \in \{rep_i[j], rep_{i,j}\}\}$, such that $rep_{i,j}$ is a message that was sent in line 102 from p_i to p_j. Moreover, $msg_{leader_{cloudID}.id}[k] = input_k$, where $p_k \in leader_{cloudID} \cup guards_{cloudID}$. (4) No counter has reached MAXINT and there are no $\langle \mathsf{RESET} \rangle$ messages.

We say that an execution is *fair* if every step that is applicable infinitely often is executed infinitely often. Theorem 5.1 demonstrates self-stabilization and uses the term *(asynchronous) cycles* of a fair execution R. A cycle is the shortest prefix of R in which every non-failing node p_i performs a completed iteration of node p_i's do forever loop, all messages that p_i sent during that iteration were delivered, and all of the iteration's requests were replied.

Theorem 5.1: The system's state is safe within $\mathcal{O}(1)$ cycles.

Proof sketch. The proof considers the predicate $pred = \{\bot\} \neq (\{info\} \cup \{infoAck[k] : k \in C \setminus suspectedCloudlet_{cloudID}(C)\})$. First consider executions in which $pred$ holds, *i.e.,* $p_{cloudID}$ does not execute lines 14–15. Under this assumption, we show that

Algorithm 2: Code for IoT iot_i

```
19 Local state: $model$: a data structure that encodes the recent sensory readings; $cloudletModel$: recent model received from the cloudlet; $cloudletList$: a list (bounded by $cloudletListSize$) of dissemination points (ordered by descending priority); $lastUpdate$: time of the last update reception from a cloudlet (according to IoT's local time); $msgseq$: a positive integer used as a sequence number for messages sent to cloudlets; $MSG$: a set of $(id, seq)$ pairs that stores the highest message sequence received by cloudlet $id$;
20 Interface: $update()$: receives the last sent $model$ and received $cloudletModel$ as well as the time in which that reception occurred ($lastUpdate$). The function then updates $model$ (and returns true) if the cloudlet model requires an update due to change in sensory input, a timeout due to a missing acknowledgment from the cloudlet or a change in the cloudlet model specifications;
21 Function: $iotInit()$: the IoT device first resets all variables dealing with Cloudlet data and control information as well as local data and control variables. Then it sends a special message $INIT$ to the Cloud, so that the Cloud removes all information about this device from the Cloudlets. Once this is done, the Cloud returns an acknowledgment to the device, and the function returns.
22 do forever /* use predefined periodicity */ begin
23     if $(clock() - lastUpdate) > LIMIT$ then $IoTinit()$;
           send($cloudID, \langle \mathsf{REGISTER} \rangle$);
24     else if $update(model, cloudletModel, lastUpdate)$ then
25         foreach $id \in cloudletList$ do send($id, \langle msgseq, model \rangle$);
26         $msgseq \leftarrow msgseq + 1$ /* if a message was sent */
27         if $msgseq = MAXINT$ then send($cloudID, \langle \mathsf{RESET} \rangle$);
28 upon $m = \langle seq, list, model \rangle$ arrival from cloudlet $j$ at time $t = clock()$ begin
29     if $m.seq > MSG|j.seq$ then
30         $(\langle cloudletList, cloudletModel \rangle, lastUpdate) \leftarrow (\langle m.list, m.model \rangle, t)$;
31         $MSG \leftarrow (MSG \setminus \{(k, \bullet) : k \notin cloudletList \vee k = j\}) \cup (j, m.seq)$;
32     send($j, \langle MSG|j.seq \rangle$);
33 upon message $m = \langle seq \rangle$ arrival from cloudlet $z$ do
       $msgseq \leftarrow \max\{m.seq, msgseq\}$;
```

items 1 to 4 of Definition 5.1 hold within $\mathcal{O}(1)$ cycles. Otherwise, we show that *pred* holds within $\mathcal{O}(1)$ cycles.

Item 1. Let $(p_i, p_j) \in C \times S$. Within $\mathcal{O}(1)$ cycles, $p_{cloudID}$ updates $info$'s $devices$ and $cloudlets$ (line 13), p_i reads $devices$ and $cloudlets$ (line 47) and send to IoT p_j (line 55) $\langle \bullet, cloudletList(j, lCloudlets), m \rangle : (j, m_j) \in myIoT(devices_{cloudID}, cloudlets_{cloudID})$. Upon its arrival, p_j stores it in $cloudletList_j$ and $cloudletModel_j$ as well as updates $lastUpdate_j$ with the arrival time (line 30). Thus, $cloudletList_i = cloudletList(i, cloudlets_{cloudID}) \wedge (lastUpdate_i \leq clock_i())$. Lines 51 and 61 imply $devices_{cloudID} = \{(k, \bullet) \in deviceSet_j\}$ and line 66 implies $((z, t, \bullet) \in agreegateInfo_j \implies p_z \in C \wedge t \leq clock_i()) \wedge (j, \bullet, m_j) \in agreegateInfo_j$.

Item 2. Suppose that in R's starting state, Item 3 does not hold. Within $\mathcal{O}(1)$ cycles, any message containing $msgseq$, $msgtoiot$ or $msgc$ arrive to its destination p_j. Whenever p_i receives the message, p_i updates it's local values.

Item 3. Within $\mathcal{O}(1)$ cycles, $leader$ and $guards$ are set by $p_{cloudID}$ (line 13) and all cloudlets read (line 47). We show that if a new leader has been put in place (or the view has become inconsistent), then within $\mathcal{O}(1)$ cycles the leader proposes and installs a new view which includes itself and the guards. After that the leader resumes the round-base updates for maintaining the state among itself and the guards (lines 93, 97, and 81–85) and aggregates $input_k : p_k \in leader_{cloudID} \cup guards_{cloudID}$, such as $msg_{leader_{cloudID}.id}[k] = input_k$.

Algorithm 3: Code for cloudlet p_i

```
34 Local state: $deviceSet$: a set (bounded by $deviceSetSize$) of IoT devices
     and their most recently received models;
35 agreegateInfo: data structures encoding aggregated sensory information;
36 $msgc$: positive integer ordering messages sent to the leader and guards;
37 $msgtoiot$: a positive integer used for ordering messages sent to IoT devices;
38 $MSGc$: a set of $(id, seq)$ pairs that stores the highest message sequence
     received by cloudlet $id$;
39 $MSGSEQ$: a set of $(id, seq)$ pairs that stores the highest message sequence
     received by IoT $id$;
40 Shared registers: $info$ and infoAck: as in Algorithm 1;
41 Interface: $aggregate(deviceSet)$: returns aggregated IoT information;
42 $cloudletList(k, set)$: for a given IoT device $iot_k$ and a $set$ of cloudlets,
     this function returns the cloudlet list that $iot_k$ should use (prioritized in an
     descending order);
43 $myIoT()$: projection of the IoTs that are within the cloudlet's responsibility;
44 $cloudID$: the address of the Cloud;
45 Function: $cloudletInit()$: the cloudlet first resets all variables dealing with
     the data and control information of cloudlets and IoT devices as well as its
     local data and control variables. Then it broadcasts a special message $INIT$
     to all other cloudlets, and to the Cloud so that the other cloudlets remove all
     information about this cloudlet; the Cloud removes all relevant information
     about this cloudlet from the IoT devices. Once the cloudlet receives
     acknowledgments from all the cloudlets and the Cloud, the function returns.
46 do forever /* use predefined periodicity */ begin
47     let lInfo := $(lDevices, lCloudlets, lLeader, lGuards)$ :=
         read(info); write(infoAck[i], lInfo);
48     if lInfo $\neq \bot \wedge i \notin$ lCloudlets then {$cloudletInit()$;
         send($cloudID$, ⟨REGISTER⟩)};
49     else if lInfo $\neq \bot$ then
50         if $i \notin (lGuards \cup \{lLeader.id\})$ then
             (agreegateInfo, MSGc) $\leftarrow (\emptyset, \emptyset)$;
51         $deviceSet \leftarrow (deviceSet \setminus \{(k, \bullet) : k \notin lDevices\})$;
52         $MSGSEQ \leftarrow (MSGSEQ \setminus \{(k, \bullet) : k \notin deviceSet\})$;
53         $MSGc \leftarrow (MSGc \setminus \{(k, \bullet) : k \notin lCloudlets\})$;
54         let $(iotAdd, msgAdd) := (0, 0)$;
55         foreach $(j, m) \in myIoT(lDevices, lCloudlets)$ do
             {send($j$, ⟨$msgtoiot, cloudletList(j, lCloudlets), m$⟩);
             $iotAdd \leftarrow 1$};
56         foreach $j \in lGuards \cup \{lLeader.id\}$ do
             {send($j$, ⟨$msgc, aggregate()$⟩); $msgAdd \leftarrow 1$};
57         $(msgtoiot, msgc) \leftarrow$
             $(msgtoiot + iotAdd, msgc + msgAdd)$;
58         if MAXINT $\in \{msgc, msgtoiot\}$ then
               send($cloudID$, ⟨RESET⟩);
59 upon message $m = \langle seq, model \rangle$ arrival from IoT $j$ at time $t$ begin
60     if $m.seq > MSGSEQ|j.seq$ then
61         $deviceSet \leftarrow (deviceSet \setminus \{(j, \bullet)\}) \cup \{(j, t, m)\}$;
62         $MSGSEQ \leftarrow (MSGSEQ \setminus \{(j, \bullet)\}) \cup (j, m.seq)$;
63     send($j$, ⟨$MSGSEQ|j.seq$⟩);
64 upon message $m = \langle seq, aggregated \rangle$ arrival from cloudlet $z$ at time $t$
     begin
65     if $i \in lGuards \cup \{lLeader.id\} \wedge m.seq > MSGc|z.seq$ then
66         agreegateInfo $\leftarrow$ (agreegateInfo $\setminus \{(z, \bullet)\}) \cup \{(z, t, m)\}$;
67         $MSGc \leftarrow (MSGc \setminus \{(z, \bullet)\}) \cup (z, m.seq)$;
68     send($z$, ⟨$MSGc|z.seq$⟩);
69 upon message $m = \langle seq \rangle$ arrival from IoT $k$ do
     $msgtoiot \leftarrow \max\{m.seq, msgtoiot\}$
70 upon message $m = \langle seq \rangle$ arrival from cloudlet $z$ do
     $msgc \leftarrow \max\{m.seq, msgc\}$
```

Algorithm 4: Self-stabilizing replication for guards and leader, code for cloudlet p_i

```
71 Interfaces: $fetch()$ next multicast message, $apply(state, msg)$ applies the
     step $msg$ to $state$ (while producing side effects), $synchState(replica)$
     returns a replica consolidated state, $synchMsgs(replica)$ returns a
     consolidated array of last delivered messages, $failureDetector()$ returns
     a vector of processor ids, $cloudID$ returns the address of the Cloud;
72 Variables: $rep[] = \langle view = \langle ID, set \rangle$, $status \in$ {Propose, Install,
     Multicast}, $(multicast\ round\ number)\ rnd$, $(replica)\ state$, last
     delivered messages $msg[n]$ to the state machine, last fetched $input$ to the
     state machine, $propV = \langle ID, set \rangle$, recently live and connected
     component $FD\rangle$: an array of the state machine's replica, where $rep[i]$
     refers to processor $p_i$, and $rep[j]$ refers to the last arriving message from
     $p_j$ containing $p_j$'s $rep[j]$. $FD$ stores the $failureDetector()$ output,
     i.e., the set of processors that the failure detector considers as active.
     $myLeader$ stores the id of the local leader; $\bot$ if none. The $view.ID$
     (and $propV.ID$) is composed by the id and leader sequence installing the
     view, and counter $cnt$, in case the same leader installs a new view;
73 Shared registers: $info$ and $data$: as in Algorithm 1;
74 Macros: $roundProceedReady() = \{(\forall p_j \in view.set$: $rep[j].(view,$
     $status, rnd) = (view, status, rnd)) \vee ((status \neq$ Multicast$) \wedge$
     $[(\forall p_j \in propV.set : rep[j].(propV, status) = (propV,$ Propose$))$
     $\vee (\forall p_j \in propV.set : rep[j].(propV, status) = (propV,$ Install$))]\}$;
75 $coordinatePropose() = \{(state, msg, status) \leftarrow$
     $(synchState(rep), synchMsgs(rep),$ install$\}$;
76 $coordinateInstall() = \{(view, status, rnd) \leftarrow$
     $(propV,$ Multicast$, 0)\}$;
77 $roundReadyToFollow() = \{rep[myLeader].rnd = 0 \vee rnd <$
     $rep[myLeader].rnd \vee rep[myLeader].(view \neq propV)\}$;
78 $followPropose() = \{(status, propV) \leftarrow$
     $rep[myLeader](status, propV)\}$;
79 $followInstall() = \{rep[i] \leftarrow rep[myLeader]\}$;
80 $followMcastRnd()$ $\{rep[i] \leftarrow rep[myLeader]$;
     $apply(state, rep[myLeader].msg)$; $input \leftarrow fetch()$;$\}$
81 procedure $coordinateMcastRnd()$ do begin
82     $apply(state, msg)$; $input \leftarrow fetch()$;
83     foreach $p_j \in C$ do if $p_j \in view.set$ then $msg[j] \leftarrow rep[j].input$
         else $msg[j] \leftarrow \bot$;
84     write($data$, ($i$, $lLeader, rnd, rep[i].state$));
85     $rnd \leftarrow rnd + 1$; if $rnd = MAXINT$ then $view.set \leftarrow \bot$ /*
         Forces a view change in line 90/;
86 do forever /* use predefined periodicity */ begin
87     $FD \leftarrow failureDetector()$;
88     let $(lDevices, lCloudlets, lLeader, lGuards)$ := read($info$);
89     if $lLeader.id = i \wedge myLeader \neq i$ then $(status, propV,$
         $myLeader) \leftarrow$ (Propose, $\langle (lLeader, cnt = 0), FD \cap$
         $(lGuards \cup \{i\}) \rangle, i)$;
90     if $lLeader.id = i \wedge myLeader = i \wedge ((status =$ Multicast $\wedge$
         $view.set \neq S) \vee (status \neq$ Multicast $\wedge\ propV.set \neq S))$
         then $(status, propV, myLeader) \leftarrow$ (Propose, $\langle (lLeader,$
         $cnt$++$), S) \rangle, i)$, where $S := FD \cap (lGuards \cup \{i\}$;
91     if $k \neq i \wedge i \in lGuards \wedge k \in FD$, where $k = lLeader.id$ then
         $(myLeader, status) \leftarrow (k, rep[k].status)$;
92     if $lLeader.id = i \wedge roundProceedReady()$ then
93         if $status =$ Multicast then $coordinateMcastRnd()$;
94         else if $status =$ Install then $coordinateInstall()$;
95         else if $status =$ Propose then $coordinatePropose()$;
96     else if $lLeader.id \neq i \wedge i \in lGuards \wedge lLeader.id \in$
         $FD \wedge roundReadyToFollow()$ then
97         if $status =$ Multicast then $followMcastRnd()$;
98         else if $status =$ Install then $followInstall()$;
99         else if $status =$ Propose then $followPropose()$;
100    if $lLeader.id \neq i \wedge i \in lGuards \wedge lLeader.id \notin FD$ then
         $myLeader \leftarrow \bot$; write($data$, ($i$, $lLeader, rnd, rep[i].state$));
101    else if $myLeader \neq \bot$ then send $rep[i]$ to $myLeader$;
102    if $lLeader.id = i$ then $\forall k \in lGuards \cap FD$ send $\langle rep[i] \rangle$ to $p_k$;
103    if $cnt = MAXINT$ then send($cloudID$, ⟨RESET⟩);
104 upon message $m$ arrival from $p_j$ do $rep[j] \leftarrow m$;
```

Item 4. Suppose that in R's starting state, Item 3 does not hold at node p_i. Within $\mathcal{O}(1)$ cycles, either Item 3 holds and Item 4 does not, or both hold. Moreover, $\langle \mathsf{RESET} \rangle$ arrives to $p_{cloudID}$ and the assumption above does not hold (line 18).

For the case that is competently to the assumption in the proof beginning, suppose that any prefix R' of $R = R' \circ R''$ that has $\mathcal{O}(1)$ cycles, does not have a matching suffix R'' during which the predicate $pred$ holds. Since $pred$ does not hold during R', $p_{cloudID}$ does not execute lines 8–13 during R'. Therefore, it must execute either line 14 or 15 for a constant number of times during R'. Suppose that $p_{cloudID}$ does not execute line 14 during R'. Thus, within $\mathcal{O}(1)$ cycles, $p_{cloudID}$ executes repeatedly line 15 until the if-statement condition of line 14 holds. Then, the if-statement condition of line 15 does not hold again during R, and, within $\mathcal{O}(1)$ cycles, the if-statement condition of line 7 holds. ■

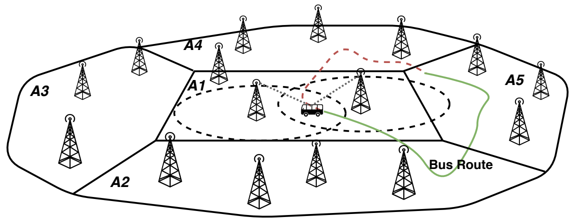

Fig. 4. Bus Network Service with 16 base stations covering 5 areas of a city

VI. Evaluation

The previous section details the correctness proof of our self-stabilizing algorithmic process which shows that *even in the presence of failures a fog ecosystem can always recover and correctly compute analytic insights from IoT data*. Most importantly, *this can be achieved in a constant number of communication rounds, in contrast to the current state-of-the-art in self-stabilization for IoT systems, where fault recovery is bounded, at best, by the number of participating entities*.

This section introduces a thorough evaluation of our framework effectiveness and runtime overhead. First, we measure *information delay*, the time interval required for IoT data to be propagated in the network for analytics to be correctly derived in the presence of multiple and different failures (*e.g.,* cloudlet fail-stop, link drops). Second, we measure the additional *runtime footprint* that our framework incurs to exemplary state-of-the-art distributed control planes (*e.g.,* istio). This provides a detailed overview of what is the cost, in terms of network overhead, of maintaining *data freshness* and analytic computation *correctness* in the presence of failures. *Results show that with our self-stabilizing framework, control planes are able to compute analytics correctly with the information delay maintained relatively stable despite of the presence of failures, while the network overhead scales linearly towards the IoT load, as required by O1-O4 (Section III).*

As a testbed, we introduce a real-world use-case of a Bus Network Service (BNS). We opt to focus on experiments that use a publicly available and real-world workload to reveal the strengths of our framework. Specifically, the workload originates from the Dublin BNS [23], comprised of 40GB of compressed data, tracking for 1 month the bus routes of 968 buses (Jan. 2013). Each bus is equipped with a GPS tracker recording every $1s$ location coordinates and the current bus route delay. Figure 4 depicts a high-level overview of the BNS topology, where 16 cloudlets are deployed across Dublin's major city regions, denoted for clarity as A_x, to decentralize the BNS and increase the system responsiveness. A bus route may span across city regions and a bus can be connected to multiple cloudlets depending on the cloudlet coverage. Each cloudlet serves as an analytics engine that aggregates local bus updates and propagates an alert to traffic operators (central cloud service) when 10 or more buses in a city area are reporting, in a 5min sliding window, delays over one standard deviation from the previous weekly mean.

To experiment with large-scale deployments and ensure both result reproducability and algorithm adoption, we have designed a testbed inspired by Kompics [24], an open-source distributed systems message-passing simulator, and extended the entity behavior model to support our self-stabilizing control plane for fog ecosystems and to facilitate complex fault models. The testbed is run in an Openstack cloud configured with 16VCPU clocked at 2.66GHz, 16GB RAM and 260GB disk. The network configuration between testbed entities adopts a gaussian kernel with 1 standard deviation and the following mean values: (i) Cloudlet-to-Cloud latency 100ms; (ii) IoT-to-Cloud latency 250ms; (iii) intra-region Cloudlet-to-Cloudlet latency 10ms; (iv) inter-region Cloudlet-to-Cloudlet latency 100ms; and (v) IoT-to-Cloudlet latency 20ms. We opt for these specific capabilities so that the testbed resembles an actual geo-distributed fog deployment over a city environment. All simulation scenarios are run 100 times with cloudlets and IoT devices starting at randomized time intervals. For the IoT device placement, we have implemented the registration interface of Algorithm 2 so that when an IoT device (*e.g.,* a bus) requests to join the network, the central authority (*e.g.,* the cloud) responds with a list of valid cloudlets that are the "closest" in the device's operating (city) region. The same strategy will hold for when the device has changed it's operating region (*e.g.,* bus moves from A_x to A_y). Finally, the selection of the leader and the guards was done randomly, since our cloudlets are homogeneous. For the widespread experimentation of different fault scenarios, we adopt the Netflix Chaos Monkey framework [25]. This enables the configuration and (random) selection of faults and entities to infest at given time intervals, or at random, depending on the evaluation scenario. Unless otherwise stated, the aforementioned topology and network configuration will be considered as the ***baseline configuration***.

A. Information Delay

In this set of experiments, we show the effect of different failures to the timeliness of analytic computation. We consider four experiment runs with faults injected at random and examine how information delay is affected by: (i) randomly failing a different number of regular cloudlets; (ii) failing the guards; (iii) failing the leader; (iv) randomly dropping the communication link between IoT devices and cloudlets.

Figure 5 depicts the information delay as the number of concurrently failing cloudlets increases. In this box-plot the median information delay is denoted by the line in the box, while the box length extends between the first and third quantile with outliers depicted as independent points. With zero cloudlets we denote the information delay in normal operation (without failures). From Figure 5, we observe that while the number of failing cloudlets remains under 7, information delay is not affected, despite slight deviations. After this, randomly selecting concurrent cloudlets hinders the extreme case of wiping out all cloudlets of a city region. This results in added delay as IoT data for the specific region must be directly propagated to the cloud. For this experiment, system recovery is only required when an IoT device is left with no cloudlet in coverage. In this extreme case, the IoT device must contact the cloud to validate the registration. However, the involvement of

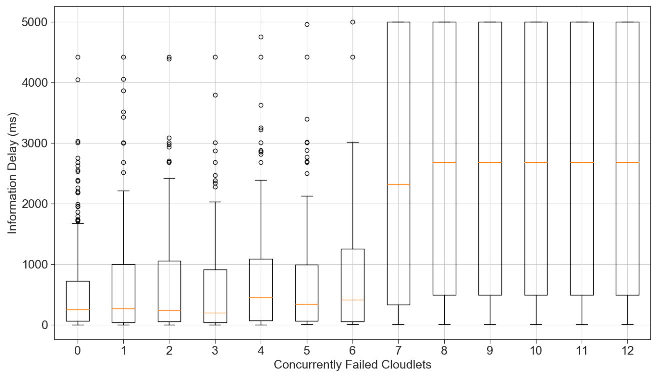

Fig. 5. Delay vs number of concurrent cloudlet failures

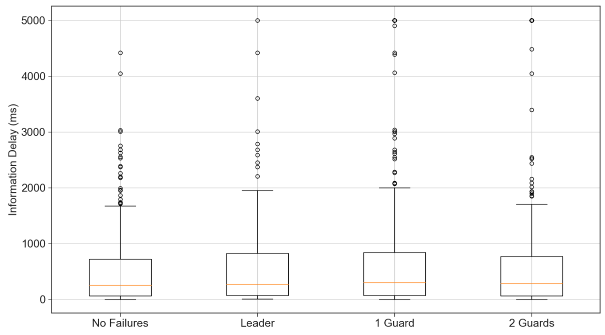

Fig. 6. Delay vs number of concurrent guard and leader failures

the cloud naturally hinders a communication overhead. Thus, *despite information delays for extreme cases of concurrent cloudlet failures, analytics computation is always correct while the system recovers from faults in a bounded number of communication rounds, as required by O1 and O3 (Section III).*

Figure 6 depicts how information delay is affected by the failure of the control plane when the baseline is configured with two guards. We observe that the timeliness of analytics computation is not affected by their failure. This concurs with the correctness proof that shows that, *the self-stabilizing fog ecosystem can return back to a legal state within $\mathcal{O}(1)$ time, which is sufficient to propagate information without delay, as required by O4 (Section III).*

The next experiment studies how information delay is affected by the temporary drop of the network link between IoT devices and cloudlets. To achieve this, we artificially block for a predefined interval the link between affected IoTs and cloudlets in each region, thus maintaining only the link with the cloud. In Figure 7 we observe that the information delay increases as more devices experience a link drop. This occurs because the affected IoTs detect the link absence and, thus, must communicate with the cloud for updates which takes more time. Still, *analytics are computed without corrupted or missing IoT data*. This extreme case, of failing all the communication links among IoT and cloudlets, highlights the importance of having a sufficient amount of cloudlets in each region to cope with concurrent link failures.

B. Runtime Footprint

In this set of experiments, we provide an analysis depicting the network overhead of different components comprising

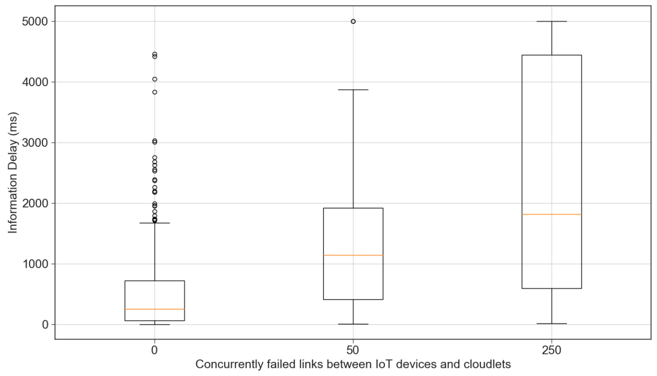

Fig. 7. Information delay vs concurrent fail-stop IoT-Cloudlet network links

our framework. Figure 8 depicts the network traffic over the data and control plane for a simulation run of the baseline configuration when random failures of the cloudlets' leader, guards, and cloudlets are introduced. The figure depicts the network overhead for 5min where a $30s$ bootstrap period is omitted. First, we observe four distinct segments (separated by vertical lines). During each segment our framework maintains a stable message exchange rate for both planes, with the data plane traffic approximately x3.5 higher than the control plane traffic. In the first segment ($30s$ to $75s$) the system exhibits no faults (baseline). At the 75th second, the leader fails and we observe a slight drop in both the control plane traffic (from 950KB/s to 850KB/s) and the data plane (from 3300KB/s to 3100KB/s). When the cloud discovers the leader failure, it elects a new leader at the 88th second and the system recovers back to a legal state, with a slight increase of the control plane traffic (900KB/s). Next, at the 150th second both guards fail and the traffic falls to 700KB/s and 2600KB/s for the control and data plane respectively. As before, the cloud elects new guards and the control plane traffic stabilizes at 750KB/s. Finally, at the 225th second (4th segment) three cloudlets fail and both control and data traffic drop to 550KB/s and 2100KB/s, respectively. These results show that a constant number of messages is exchanged, validating the objectives O1 and O3.

Next, we show that the control and data plane network traffic scales linearly, as required by O2. Table I shows the results of different configurations in percentage increments from the baseline configuration.

Guards. We observe that the overhead of adding guards increases linearly. Specifically, each additional guard adds an overhead in the range of $4.75 - 5.68\%$ for the control plane traffic and $4.82 - 5.02\%$ for the data plane traffic. It is worth pointing out that the previous experiment in Figure 6 showed that even with all the guards failing concurrently, the information latency remains stable, and therefore, for the studied baseline configuration, having two guards balances well the trade-off between overhead and information delay.

Cloudlets. The overhead of adding extra cloudlets, for redundancy purposes, scales linearly while the IoT load remains stable. Specifically, each additional cloudlet adds an overhead in the range of $7.96 - 9.03\%$ for the control plane, and for the

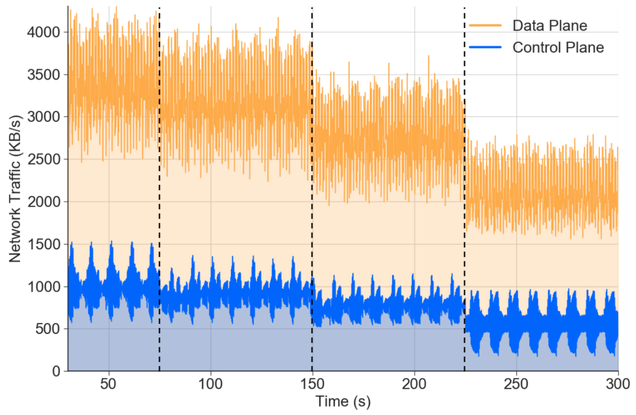

Fig. 8. Control and Data Plane Traffic

System Change	Control Plane Traffic Change Compared to Baseline (%)	Data Plane Traffic Change Compared to Baseline (%)
3 Guards	4.75	5.02
4 Guards	10.38	9.64
5 Guards	17.04	14.50
6 Guards	22.06	19.44
20 Cloudlets	31.85	25.03
25 Cloudlets	76.63	56.38
30 Cloudlets	126.48	87.85
1500 IoT	46.76	43.25
2000 IoT	93.59	85.58
2500 IoT	140.18	127.51

TABLE I
NETWORK TRAFFIC OVERHEAD OVER TOPOLOGY CHANGES

data plane the increment is approximately 6.2%. Obviously, the trade-off is straightforward. Increasing the cloudlets, decreases the probability of delaying information propagation (*e.g.,* as in the case of Figure 5 after 7 concurrent cloudlet failures) at the cost of higher network traffic.

IoTs. By increasing the workload (IoT devices), again, the network overhead is linearly increased. Each additional IoT device adds a 0.094% overhead on the control plane traffic, while for the data plane the increment ranges between $0.085 - 0.087\%$. This increase is attributed to the fact that each cloudlet communicates with more IoT devices.

VII. Conclusions

This paper introduces a fault-tolerant framework for distributed control planes that enables fog services to cope with a very broad fault model. To this end, we presented self-stabilizing algorithms that guarantee automatic recovery within a constant number of communication rounds without the need for external (human) intervention. Using real-world data and actual queries of interest from an intelligent transportation service, we demonstrate the performance gains of our framework, and thus the promise of self-stabilization in fog computing. Our results show that despite information delays for extreme cases of concurrent cloudlet failures, analytic computation is correct, while the network overhead is proportional to the number of cloudlets, guards, and devices. We believe that our self-stabilizing framework is applicable to a wide range of fog services requiring strong fault-tolerance guarantees.

Acknowledgement. This work is partially supported by the EU Commission through RAINBOW 871403 (ICT-15-2019-2020) project and by VINNOVA (FFI) AutoSPADA project (DNR 2019-05884).

References

[1] W. Shi and S. Dustdar, "The promise of edge computing," *Computer*, vol. 49, no. 5, pp. 78–81, 2016.
[2] D. Trihinas, G. Pallis, and M. Dikaiakos, "ADMin: adaptive monitoring dissemination for the internet of things," in *IEEE INFOCOM*, 2017.
[3] L. Bittencariou, R. Immich, R. Sakellariou, N. Fonseca, E. Madeira, M. Curado, L. Villas, L. DaSilva, C. Lee, and O. Rana, "The internet of things, fog and cloud continuum: Integration and challenges," *Internet of Things*, vol. 3-4, pp. 134 – 155, 2018.
[4] T. He, H. Khamfroush, S. Wang, T. La Porta, and S. Stein, "It's hard to share: Joint service placement and request scheduling in edge clouds with sharable and non-sharable resources," in *ICDCS*, 2018, pp. 365–375.
[5] D. Trihinas, G. Pallis, and M. Dikaiakos, "Low-cost adaptive monitoring techniques for the internet of things," *IEEE Transactions on Services Computing*, pp. 1–1, 2018.
[6] D. Trihinas, L. Chiroque, G. Pallis, A. Fernandez, and M. Dikaiakos, "ATMoN: Adapting the "Temporality" in Large-Scale Dynamic Networks," in *38th IEEE Distributed Computing Systems (ICDCS)*, 2018.
[7] A. Alarifi, F. Abdelsamie, and M. Amoon, "A fault-tolerant aware scheduling method for fog-cloud environments," *PLOS ONE*, vol. 14, no. 10, pp. 1–24, 10 2019.
[8] S. Dolev, *Self-Stabilization*. MIT Press, 2000.
[9] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Commun. ACM*, vol. 17, no. 11, pp. 643–644, 1974.
[10] Z. Georgiou, C. Georgiou, G. Pallis, E. M. Schiller, and D. Trihinas, "A self-stabilizing control plane for the edge and fog ecosystems," *CoRR*, vol. abs/2011.02190, 2020.
[11] Y. Harchol, A. Mushtaq, J. McCauley, A. Panda, and S. Shenker, "CESSNA: Resilient edge-computing," in *Proceedings of the 2018 Workshop on Mobile Edge Communications*. ACM, 2018, pp. 1–6.
[12] C. Wang, C. Gill, and C. Lu, "Frame: Fault tolerant and real-time messaging for edge computing," in *ICDCS*, 2019, pp. 976–986.
[13] K. Wang, Y. Shao, L. Xie, J. Wu, and S. Guo, "Adaptive and fault-tolerant data processing in healthcare iot based on fog computing," *IEEE Transactions on Network Science and Engineering*, pp. 1–1, 2019.
[14] V. Karagiannis, S. Schulte, J. Leitão, and N. Preguiça, "Enabling fog computing using self-organizing compute nodes," in *3rd IEEE Fog and Edge Computing (ICFEC)*, 2019.
[15] "Istio." [Online]. Available: https://github.com/istio/istio
[16] "Linkerd." [Online]. Available: https://github.com/linkerd/linkerd
[17] G. Siegemund and V. Turau, "A self-stabilizing publish/subscribe middleware for iot applications," *ACM Trans. Cyber-Phys. Syst.*, vol. 2, no. 2, pp. 12:1–12:26, Jun. 2018.
[18] M. Canini, I. Salem, L. Schiff, E. M. Schiller, and S. Schmid, "Renaissance: A self-stabilizing distributed SDN control plane," in *ICDCS*, 2018, pp. 233–243.
[19] S. Chattopadhyay, S. Chatterjee, S. Nandi, and S. Chakraborty, "Aloe: An elastic auto-scaled and self-stabilized orchestration framework for iot applications," in *IEEE INFOCOM*, 2019, pp. 802–810.
[20] S. Dolev, C. Georgiou, I. Marcoullis, and E. M. Schiller, "Practically-self-stabilizing virtual synchrony," *J. Comput. Syst. Sci.*, vol. 96, pp. 50–73, 2018.
[21] A. Binun, M. Bloch, S. Dolev, R. M. Kahil, B. Menuhin, R. Yagel, T. Coupaye, M. Lacoste, and A. Wailly, "Self-stabilizing virtual machine hypervisor architecture for resilient cloud," in *2014 IEEE World Congress on Services*, 2014, pp. 200–207.
[22] P. Blanchard, S. Dolev, J. Beauquier, and S. Delaët, "Practically self-stabilizing paxos replicated state-machine," in *Networked Systems NETYS*, 2014, pp. 99–121.
[23] "Dublin smart city bus network data." [Online]. Available: https://data.smartdublin.ie/
[24] C. Arad, J. Dowling, and S. Haridi, "Message-passing concurrency for scalable, stateful, reconfigurable middleware," in *Middleware 2012*. Springer Berlin Heidelberg, 2012, pp. 208–228.
[25] "Chaos monkey tool." [Online]. Available: https://github.com/Netflix/chaosmonkey