Overlay Networks for Edge Management *

Pedro Á. Costa[1], Pedro Fouto[1], and João Leitão[1]

[1]*NOVA LINCS & DI/FCT/NOVA University of Lisbon, Lisbon, Portugal,*
{pah.costa, p.fouto}@campus.fct.unl jc.leitao@fct.unl

Abstract

Edge computing has emerged as a solution to address existing limitations of cloud computing for bandwidth-heavy and time-sensitive applications, by moving (some) computations from bandwidth saturated Cloud infrastructures closer to client devices, where data is effectively produced and consumed. However, existing materializations of the edge computing paradigm take limited advantage of computational and storage power that exists in the edge and between client devices and the cloud. Most of these leverage static hierarchical topologies (e.g., Fog Computing) to pre-process data before sending it to the Cloud, which limits the advantages that can be extracted from the edge computing paradigm. In the past, peer-to-peer systems have sought to tackle the challenges of increasing scalability and availability for very large systems, with a large number of solutions being proposed namely, distributed overlay networks for resource management. In this paper, we argue that the clever adaptation of peer-to-peer solutions can enable novel applications to fully exploit the potential of the edge. In particular, we study the viability of taking advantage of specialized overlay networks in edge environments to enable the management of a large number of computational resources. Contrary to previous proposals, that assume the environment to be composed of mostly homogeneous devices, our proposal embraces existing heterogeneity and exploits the location of computational resources to devise a (partially) self-organizing overlay network that can be exploited both to provide membership information to applications, but also do efficiently disseminate management information across edge devices. We have conducted an experimental evaluation using container-based emulation in an heterogeneous network composed by 100 devices, with results showing that our protocol is able to maximize the bandwidth usage of the system, allowing more data to flow throughout the network, while retaining high robustness to failures.

*Parts of this report are published as part of the Proceedings of the 19th IEEE International Symposium on Network Computing and Applications (NCA 2020). The work presented here was partially supported by FC&T through NOVA LINCS (grant UID/CEC/04516/2013) and NG-STORAGE (PTDC/CCI-INF/32038/2017). Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

1 Introduction

Today's cloud-based applications are subjected to ever-growing volumes of data produced by an increasing number of clients (either user or IoT devices), which require fast, available, and reliable responses [3]. Unfortunately, this trend will induce a prohibitively high load on network infrastructures that connects clients to cloud data centers, whose available bandwidth is not increasing at the same rate as the load induced on them. The edge computing paradigm [20] emerged to address these limitations by proposing to move (some) computations from the center of the network (i.e., the cloud) towards clients (i.e., the edge).

Edge computing consists in performing computations outside of cloud infrastructures in computational resources that are closer to data producers and consumers. However, edge computing can have many different materializations due to its broad definition [9]. One of the most popular is fog computing [1, 15, 25], which promotes the placement of specialized servers in close vicinity to client devices, promoting a simple hierarchical topology where fog servers pre-process data before shipping it to the cloud, thus being able to provide timely responses to some of the client requests without waiting from the cloud. This simple 3-tier (static) architecture [23, 28], with the cloud data centers at the top, a fog server in the middle, and client devices that produce and consume data at the bottom, presents several limitations in terms of flexibility, availability, and reliability, as the unavailability of a fog server can easily disrupt the operation of a system in a large area, since each fog server typically handles multiple client devices in a given geographic location. Addressing this can easily incur in high cost, either through redundant fog hardware or high amounts of manual management. These issues are more evident in smart city domains, such as connected heathcare [20] and autonomous traffic systems [2], where such failures can lead to inadmissible unavailability and disastrous results.

For such critical application domains, as well as applications that mediate interactions among human participants [9] that require constant availability and fast response times, edge computing infrastructures should evolve to be more dynamic and flexible, promoting decentralized self-management and self-healing capabilities as to ensure high availability and reliability with minimal operational costs. This might entail taking advantage of other devices with computation power that exist in the path between client devices and core cloud infrastructures. Furthermore, we argue that the needs of such applications often require geographical proximity of edge resources to be considered, as to promote processing that regards the location of clients (and the data produced and consumed by them) as well as to lower response times.

The challenges of managing large scale systems in a (mostly) decentralized way have been previously tackled in the context of peer-to-peer (p2p) systems [17], where various solutions emerged, namely distributed protocols that maintain the system membership by building and managing a self-healing decentralized overlay network [8], which can adapt themselves to the needs of the applications operating on top of them. In this paper we posit that such techniques can serve as the foundation of novel architectures that can allow edge computing solutions to achieve the properties discussed above, paving the way for novel edge-enabled applications to emerge [9], in a hybrid cloud-edge environment that takes full advantage of computational resources that span from the cloud to the edge.

To this end, we explore how distributed overlay protocols can be leveraged to provide nodes with (partial) membership information about resources in close vicinity, and how such topologies can be used to efficiently propagate management information that can be leveraged to support the operation of edge-enabled applications (in this context such information can either be monitoring information [13] or management commands [14]). Furthermore, we note that the continuum of resources in the edge is naturally heterogeneous, exhibiting highly variable capabilities in terms of processing capacity and available bandwidth. Therefore, we strive to ensure that nodes with more capabilities are positioned in the overlay in a way that such nodes contribute more to both managing the overlay topology and disseminate control information. We achieve this with minimal configuration information, requiring system administrators to configure each node with only the contact information of an entry point in the system and a numerical *level* that intuitively captures the proximity of the node to cloud data centers, whereas we assume that nodes closer to the core of the network have more computational resources and available bandwidth [9, 20, 25].

In summary, in this paper we make the following main contributions: *i*) we propose *Bias Layered Tree*, a novel decentralized overlay network that builds a robust hierarchical tree-based topology connecting the cloud infrastructure and edge devices, in a way that takes into account their level and relative network proximity; *ii*) we provide a reference implementation of Bias Layered Tree and several existing p2p overlay management and dissemination schemes in a unified code base[1]; *iii*) we conduct an extensive experimental study on the impact of different overlay protocols and data dissemination mechanisms through emulation, using an heterogeneous network composed of 100 devices, which as far as we know is among the first experimental studies using real code in a realistic heterogeneous environment.

The remainder of this paper is organized as follows: § 2 discusses the requirements of management infrastructures for hybrid cloud-edge environments; § 3 discusses the state of the art, providing a survey on popular distributed protocols for building overlay networks; § 4 presents the design of Bias Layered Tree; § 5 details our experimental work; and finally, § 6 concludes the paper.

2 Requirements for Edge Management

To allow applications and systems to take full advantage of the edge computing paradigm, it is essential to understand the key properties of the hybrid cloud-edge setting. Many authors have proposed models for the deployment and execution of applications across the cloud and the edge [4, 9, 20, 25, 26]. In this paper, we define the cloud-edge environment as being composed of a set of devices that can either be the cloud or located in between the cloud and client devices (including the extremes). Examples of edge devices include mini or regional data centers, points-of-presence, ISP servers, private servers, 5G towers with computing capabilities, among others. We do not make any assumptions regarding the types of devices, only that each has its own CPU, memory, and network connection. We assume that each device can have associated with it a number that we denominate *level* that captures how far it is from the cloud infrastructure (i.e., cloud devices have a level of zero, whereas the level increases as we get closer to client devices). We also assume that CPU, memory, and network capabilities of devices tend to decrease as the level increases. We assume that

[1]Available at `https://github.com/pedroAkos/EdgeOverlayNetworks`.

system administrators that configure devices to be part of a distributed cloud-edge platform can provide an adequate level to each device considering both its location and available resources.

The inherent complexity of this environment justifies the need of specialized infrastructure support for applications that aim at exploiting the full range of possibilities of the hybrid cloud-edge environment. We further consider that solutions with this aim should strive to have the following properties:

Decentralized

The cloud-edge environment is composed by a large number of computational devices that exist between the cloud and client devices themselves. This translates into a vast number of computational resources, potentially under different administrative domains, that require some form of cooperation and coordination mechanisms to allow applications to take advantage of them through robust services. A solution that relies solely on central authorities to manage the system will be unable to efficiently manage such large number devices, not only due to the scale of such infrastructures, but also because that would become a lock-in point for regional providers. This implies that control and management should be decentralized enabling computational platforms operating in the cloud-edge environment to scale and evolve in an organic and efficient fashion.

Autonomous Fault Tolerance

A system that operates in the cloud-edge environment will naturally be subjected to a high level of dynamism in the form of nodes becoming available and unavailable (a phenomenon usually called *churn* [22]). This can be a consequence of, in addition to device failures, independent administrative decisions, consumption and potential exhaustion of resources of edge devices, and network anomalies (such as transient network partitions). This implies that platforms to support the operation of applications in such an ecosystem have to be designed to naturally self-reconfigure in the presence of failures or other anomalies. Furthermore, its operation should not require human intervention as to minimize operational costs and avoid human error.

Maximize Proximity & Minimize Latency

One of the key goals of edge computing is to reduce the latency between centralized services and their clients, by moving key components of these services to locations closer to clients. This implies that an efficient management solution should take into consideration the relative proximity between resources being managed. To this end, one can take into consideration the latency between devices, which can be measured in the background. However, this introduces a constant network cost to keep measurements up-to-date while, at the same time, measured latency can be highly variable. In this paper we opt to capture the notion of proximity through the use of an heuristic: the (bit-wise) length of the common prefix between two IP addresses. This metric has been shown previously to have a relevant correlation with network proximity among devices [7], where larger common prefixes translates to nodes in closer vicinity. This heuristic provides stable distance metrics and can be computed by nodes without resorting to the exchange of messages.

Adequate Load Distribution

As discussed previously, edge resources are highly heterogeneous when considering their capacity in terms of processing, memory, and network capabilities. Due to this, it is relevant to ensure that the load imposed on devices is distributed in a way that respects their capabilities, as to benefit applications without overloading them. One way to achieve this is to ensure that resources are logically organized in a way that takes their different capabilities into consideration as to ease the process of distributing tasks to them. Furthermore, the contribution of each device to maintain the management infrastructure covering the cloud-edge spectrum should also take this factor in consideration, where more powerful devices should contribute more.

Most of the requirements discussed above have been also pursued in the context of peer-to-peer (p2p) systems, particularly in the context of overlay networks and data dissemination primitives, which provide abstractions that are essential to the operation of a management platform for cloud-edge environments. This motivates us to study the applicability of some of these ideas, and how to evolve them, towards building infrastructure support for edge-enabled applications.

3 Related Work

In this work we focus in exploring decentralized overlay network protocols to manage a set of heterogenous resources across the cloud-edge environment with the goal of providing to applications a platform that allows them to reap the potential of edge computing. In the following we discuss some of the most relevant solutions in the literature for building and managing overlay networks, that typically are classified as being structured or unstructured. We also discuss other solutions that take advantage of such topologies to manage resources and disseminate information. Some of the most illustrative proposals from the literature are used in our experimental work as baselines to our own proposal.

Structured Overlays Structured overlays are mostly employed for decentralized and efficient resource look-up and application-level routing. To achieve this, most solutions, such as Chord [21], Pastry [18], and Kademlia [16], are based on a Distributed Hash Table, that typically organizes nodes in a topology that is known a-priori based on node identifiers, such as a ring. Overall, structured overlays have been shown to be easily affected by churn, since failure leads nodes to have errors in their tables, which are reinforced by the (concurrent) arrival of new nodes, which ultimately leads the topology to break with small chances of ever recovering. Furthermore, to ensure the invariants associated with the target topology, when a node is suspected of being faulty, the process to replace that node in the topology typically requires additional time, which allows for topology errors to be reinforced. Additionally, most structured overlay protocols assume all peers to have similar capabilities, and do not support hierarchical organization of nodes which we employ in our own proposal.

Unstructured Overlays Unstructured overlays, also referred as random overlays, are commonly employed to build scalable and robust membership protocols/services. They provide an alternative to structured overlays since the partial views maintained by each node have few or no invariants that have to be enforced, hence the overlay topology can be adjusted in response to failures in a timely manner. Protocols that build unstructured overlays are usually distinguished by the strategies employed to maintain their partial views, which are key to guaranteeing connectivity (avoiding partitions and node isolation) and minimizing path lengths (and consequently overlay diameter). These strategies can be either reactive, where changes to partial views are only performed in reaction to some external event (e.g., a suspected fault, reception of a message, etc); or cyclic, where partial views are periodically updated. Reactive strategies allow for more stable topologies, while cyclic strategies provide better sampling quality over the nodes that are part of the system [8].

Cyclon [27] is a peer sampling protocol that builds a continually changing overlay network optimized to perform gossip dissemination. To this end, Cyclon leverages a fixed length partial view which is maintained by a cyclic strategy where nodes periodically exchange messages containing samples of their partial view which are used to update their local view of the system. Cyclon associates with each node identifier an age value (i.e., monotonic counters that are incremented only by the local node) which allows these identifiers to be tested in the periodic exchange of Cyclon, ensuring that faulty nodes are eventually removed from every correct node partial view.

HyParView [12] combines a reactive strategy with a cyclic strategy, maintaining two partial views (one managed by each strategy). Due to this, HyParView is able to maintain a stable small (symmetric) active view using a reactive strategy, which effectively forms the overlay network used to support communication among nodes; and a larger passive view, maintained by a cyclic strategy, that is used to replace suspected nodes from the active view. To effectively maintain the passive view, HyParView performs random walks to exchange samples of both the active and passive views. This allows nodes to obtain randomized samples, while only communicating through the connections associated with the active view.

Although unstructured overlays tend to be very robust to high levels of dynamism, this comes at the cost of not being optimized to a specific goal, such as minimizing latency, which is a desired property of fog architectures. However, there are solutions that adapt the topology of overlays to improve application-level requirements.

Biased and Optimized Overlays Several work have explored how to improve the topology of overlay networks considering a given criteria. Similarly in this work, we build a specialized overlay network whose topology that maps to the relative position of nodes in regard to a data center and among themselves. MON [14] takes advantage of a gossip-based dissemination mechanism to embed a short-lived dissemination tree among nodes which can be used to disseminate administrative commands efficiently. Similar to our work, MON attributes a numerical level to a node that represents the logical distance of that node to the root of a tree, avoiding the creation of cycles. Unfortunately, MON is tailored for short interactions and hence the protocols lacks any fault tolerance mechanism to recover the tree topology. Plumtree [11] uses a similar strategy to MON to embed a tree over a random overlay topology that can be used to efficiently disseminate information. Plumtree uses links of the random overlay that are not part of the tree to convey small control information that allows to detect (in a decentralized fashion) partitions in the tree and recover them locally. Plumtree assumes a random overlay over which it operates, and does not make any consideration about the capabilities of nodes, which can lead nodes with limited capability to become interior nodes in the tree, responsible for transmitting several messages, whereas more powerful nodes can become leaves, not contributing with their resources to disseminate messages. Our work builds an overlay that (as we show further ahead) can be leveraged by Plumtree to build more effective dissemination trees.

T-MAN [6] and X-BOT [10] are two protocols that iteratively adapt the neighboring relations among nodes in a random overlay to optimize a given performance criteria. T-MAN ensures that the topology improves faster but at the risk of allowing the topology to break, wheres X-BOT protects the connectivity of the overlay by leveraging slower link exchanges with minimal coordination among nodes, while ensuring that a few links are never optimized. Contrary to our solution, both of these protocols assume nodes to be similar in capacity, biasing the overlay topology by mostly considering proximity metrics (either based on network aspects such as latency or node identifiers).

4 Bias Layered Tree

In this section we describe the design of a novel decentralized overlay network that we have designed to cope with the requirements of a resource management platform to support edge computing. The protocol takes inspiration from HyParView [12] by taking advantage of two complementary partial views: active and passive. Similar to HyParView, the active view contains information on peers with whom the local node interacts, and the passive view contains information on candidate peers that are used to recover from faults. Contrary to HyParView, Bias Layered Tree manages the contents of the active view to build and maintain a tree topology across nodes in the system by leveraging the previously described *level* property, that encodes the distance of a given node to the cloud infrastructure as well as serving as an indicator of its capacity, and ensures a set of restrictions and properties that are not commonly found in existing unstructured overlay networks such as HyParView.

Pseudo-code notation

In the next sections we provide the specification of Bias Layered Tree which is presented as pseudo-code in Algorithm 1. In the pseudo-code each node identifier is a tuple with four fields, which encode the following relevant information: *i*) network identifier, i.e., an IP address and port; *ii*) age, which is an integer that represents how much periodic communication steps have passed since a message was exchanged with the node (which is similar to the age parameter employed in the design of Cyclon [27]); *iii*) timestamp, which is a monotonically increasing counter, which is only updated by the node identified by that tuple; and *iv*) status, which is a binary value that encodes if the local node believes that peer to be correct or suspected of failure. This last field is taken into consideration whenever ordering sets of node identifiers, ensuring that suspected nodes appear last. This allows to eventually remove node identifiers of suspected nodes from passive views. In the pseudo-code, a field with a value of _ is a wildcard for *any value*, and is used to denote when that field value is not relevant for that step in the protocol. Furthermore, function `genId` is used to return a new tuple for a given node, with its age set to zero, and a new timestamp.

4.1 Protocol State

The first few lines of Alg. 1 summarize the main parameters used to control the operation of our algorithm and the main data structures used to maintain the state of the protocol at each node.

As discussed before, the *activeView* encodes node identifiers with whom the local node exchanges information frequently. This partial view is divided in three components that have different sizes and represent the relationship of a node with its peers. The first component has at most a single identifier for the parent node, which must be on a lower level (i.e., closer to the cloud) than the local node. The second component of the active view contains a fixed amount of peers that have the same level as the local node (controlled by parameter *siblingLimit*), while we refer to such peers as siblings, they may not share the same parent. Finally, the third component of the active view is dedicated to maintaining the identifiers of children nodes, having no strict limit to its size. Peers in that component must be on a higher level (i.e., farther from the cloud) than the local node. Notice that this last component leads active views to have different sizes across nodes depending on the number of children. Neighboring relationships are symmetric between peers in different levels but not among siblings. A fault detector is locally executed for peers in the active view, we use persistent TCP connections to this end similarly to HyParView.

The *passiveView* contains information about peers across different levels that is used to recover from faults and as a source of potential candidates to improve the tree topology maintained by Bias Layered Tree. Since levels have to be taken into account when recovering from failures, in Bias Layered Tree, passive views are managed to ensure that its contents provide information about peers in relevant levels. Each node strives to have a given number of identifiers for peers in the same level (defined by parameter *passiveLimit*), and the number of identifiers for the remaining levels decreases as the distance of that level increases. E.g., if a node in level 3 is parameterized to have 5 identifiers of other nodes in its level, it will strive to have 4 identifiers for levels 2 and 4, 3 identifiers for levels 1 and 5 and so forth. The rationale for this is that most recoveries will take place by connecting to peers in proximity to the local node (where the level value encodes this abstract notion of proximity).

Notice that Bias Layered Tree strives to also bias the topology of the generated tree so that links are preferably established among nearby peers. To this end, we rely on the IP address common prefix as a distance criteria. To support this behavior, nodes keep the best peers according to that criteria in their active and passive views, by sorting these views (similar to T-Man [6]). To effectively build the tree (i.e., create the links among nodes in different levels), nodes are fully delegated to select their parent according to their local view of the system. Since the correction of the topology requires these links to be correct, nodes coordinate among them to ensure the symmetry of these links, and both ends of these links monitor their remote peer.

In the following we describe the mechanism employed by Bias Layered Tree to build and maintain a level-aware tree. In particular we discuss the join procedure that defines the initial position of a node in the tree (§ 4.2), and the mechanisms respectively used to maintain the tree (§ 4.3) and to iteratively optimize the tree topology (§ 4.4).

4.2 Building the Tree

We begin the specification of Bias Layered Tree with the join procedure. The algorithm does not show the initialization procedure due to lack of space however, each node starts by recording input parameters (including their level) and initializes each view to be an empty set. Additionally, each node setups timers to perform periodic actions (which will be explained in the following sections) and sends a join message containing its level to the contact node. We assume the contact node to belong to level 0 and act as the root of tree. We further assume that this node resides on a cloud infrastructure where it can be easily replicated through solutions that provide high availability [5]. Furthermore, new nodes set a timeout for the join procedure. When the timeout triggers, the new node restarts the join procedure if it has no nodes in its active view (increasing the value of this timeout).

When the contact node receives a JOIN message, it creates an empty FORWARDJOIN message for the new node and sends it to itself (Alg. 1 lines $3-4$). The idea of the FORWARDJOIN message is to gather information regarding the local topology that contains adequate peers for the new node, allowing the joining node to bootstrap its passive view with relevant (close) peers and effectively join the tree topology by selecting a suitable parent node. To this end, the FORWARDJOIN message is propagated across the overlay through a biased random walk, that is forwarded in each level towards the (next) best node (from the local node's active view) according to the distance criteria considering the IP address of the new node (Alg. 1 line 11). Nodes record their identifiers in a *path* set sent with the message, to ensure the message is not forwarded to nodes that have already processed it. Furthermore, the message is forwarded within a given level up to a maximum time to live (ttl) number of times (controlled by parameter RW in Alg. 1). Once the ttl expires or all (locally) known peers in the current level have been visited, the message is forwarded to the next level (reseting the ttl and path). The random walk ends when the new node receives the FORWARDJOIN message (Alg. 1 lines $7-8$). To achieve this, the protocol chooses the best node for the next level from the *topo* data structure of the FORWARDJOIN message (Alg. 1 line 20). To populate the data structure, at each step, each node adds itself and all elements of its active view that belong to the next level (Alg. 1 lines $13-14$), with the new node's identifier also being added (Alg. 1 line 19). The next level, in this context, is defined as being the minimum level between the local node's level and the new node's level (there can be gaps in nodes levels in the system, i.e., a given level might not have any node associated to it). Notice however that to avoid the size of the FORWARDJOIN to grow indefinitely, at each step the sample of the network contained within it is trimmed to ensure that there is a maximum number of node identifiers per level (with also a maximum number of levels being encoded in the message), giving preference to levels that are closer to the joining node (controlled by parameters $joinTopoLevels$ and $joinTopoNodes$ in Alg. 1).

Because the FORWARDJOIN carries information about the network topology, each node that processes the message takes advantage of this information to update its local views (both active and passive) (Alg. 1 line 6) with new information. This is specially important for the new node, as this populates its passive view, allowing it to join the tree topology by selecting a parent node. To this end, the SELECTNEWPARENT procedure (Alg. 1 line 42) is triggered which computes a set of candidate parents composed by all nodes that are known to reside in a level lower than that of the new node, across both active and passive views (the active view here is used to ensure that this procedure can also be used to optimize the tree topology as discussed further ahead). The node selects a candidate parent from the resulting set. The candidate is the node with highest level (in the resulting set) that is closest to the node considering the IP prefix distance heuristic. After this, if the candidate parent differs from the current parent (which is always true during the join procedure), the node sends a PARENT message containing its old parent information (if it had one) informing the new parent (Alg. 1 lines $48-49$). The new parent introduces the node in its active view as a child, and replies back to confirm that the operation succeeded. If the candidate parent fails to respond (within a sensible time frame), it is considered to be suspected of failure. In the next sections we discuss how the algorithm handles such scenarios.

4.3 Maintaining the Tree

To maintain the tree topology, Bias Layered Tree relies on periodic interactions among nodes to gather information about the current system configuration, and find candidate peers to recover from faults and/or perform optimizations. These interactions occur whenever a *ShuffleTimer* event is triggered (Alg. 1 lines $22-27$). The *ShuffleTimer* contains a *view* parameter that is used to choose a peer to communicate with. During initialization, the protocol sets two *ShuffleTimers*, one with high frequency (e.g., every 2 seconds) and with *view* set to the active view; and another with low frequency (e.g., every 10 seconds) and *view* set to the passive view. The rational is to strive to keep localized information up-to-date, as this is most useful to recover from faults; while occasionally exploring the system membership in remote areas (which is more useful to optimize the tree topology as discussed further ahead).

The periodic interaction begins by the local node incrementing the age value of all identifiers present in its active and passive views, and selecting the oldest node in the *view* provided by the timer (when using the passive view, and if several nodes have the same age, the one that is most distant considering the IP prefix criteria is picked). After selecting the peer for the exchange step, the local node computes a sample by selecting k_a active nodes and k_p passive nodes which are closer (via the IP prefix criteria) to the chosen peer (Alg. 1 line 26) and sends that sample and a newly generated identifier for itself (i.e., an identifier with an age of 0 and a current timestamp to reinforce that the local node is correct) in a SHUFFLE message (Alg. 1 line 27) to the chosen peer.

When a node receives a SHUFFLE message (Alg. 1 line 28), it updates its active and passive views with the updated information enclosed in the message (Alg. 1 lines 29). This is achieved in the following way. For each node identifier in the received message, if that node is contained in either the active or passive view, it updates the information for that node if, and only if, the received timestamp is higher than the locally recorded one. If the node is not contained in either view, its identifier is added to the local passive view. After this, the passive view is sorted (considering the distance criteria and the status of the node as explained before) and the end of the list is trimmed to ensure that the passive view has the target size. After this step, the node that received the SHUFFLE message computes a reply

Algorithm 1: Bias Layered Tree

```
//Parameters
    $level$ //the local node level
    $siblingLimit$ //max sibling nodes
    $passiveLimit$ //passive view base limit
    $RW$ //ttl for biased random walk within a layer
    $joinTopoLevels$ //level information to carry in join
    $joinTopoNodes$ //node information to carry in join
    $Timeout$ //timeout for join procedure
    $k_a$ //sample size from active view
    $k_p$ //sample size from passive view

Local State:
1.  $activeView$
2.  $passiveView$

3. Upon Join ($l$) from $node$ do:
4.   trigger send(ForwardJoin, $node$, $l$, $RW + 1$, {}, {}) to $self$

5. Upon Recv(ForwardJoin, $node$, $l$, $ttl$, $path$, $topo$) from $sender$ do:
6.   call updateViewsWith($topo$)
7.   if node = self then:
8.     call SelectNewParent()
9.   else:
10.    ttl $\longleftarrow$ ttl $-$ 1
11.    nextHop $\longleftarrow$ getBestNextHop($path$)
12.    nextLevel $\longleftarrow$ getNextLevel($level$, $l$)
13.    topo[$level$] $\longleftarrow$ topo[$level$] $\cup$ genId(self)
14.    topo[$nextLevel$] $\longleftarrow$ topo[$nextLevel$] $\cup$ activeView[$nextLevel$]
15.    if ttl $> 0 \wedge$ nextHop $\neq \perp$ then:
16.      path $\longleftarrow$ path $\cup$ self
17.      trigger send(ForwardJoin, $node$, $l$, $ttl$, $path$, $topo$) to $nextHop$
18.    else:
19.      topo[$l$] $\longleftarrow$ topo[$l$] $\cup$ genId(node)
20.      nextHop $\longleftarrow$ getBestNode(topo[$nextLevel$])
21.      trigger send(ForwardJoin, $node$, $l$, $RW$, {}, $topo$) to $nextHop$

22. Upon ShuffleTimer ($view$) do:
23.  call IncrementAgesOfAllNodes()
24.  (oldest,_,_,_) $\longleftarrow$ getOldest($view$)
25.  if oldest $\neq \perp$ then:
26.    sample $\longleftarrow$ computeSample($oldest$) $\cup$ genId(self)
27.    trigger send(Shuffle, $sample$) to $oldest$

28. Upon Recv(Shuffle, $sample$) from $sender$ do:
29.  call updateViewsWith($sample$)
30.  sample $\longleftarrow$ computeSample($sender$) $\cup$ genId(self)
31.  trigger send(ShuffleReply, $sample$) to $sender$

32. Upon Recv(ShuffleReply, $sample$) from $sender$ do:
33.  call updateViewsWith($sample$)

34. Upon NodeDown ($node$) do:
35.  deadNodeLevel $\longleftarrow$ $i$ : activeView[$i$] $\neq \perp \wedge$ (node,_,_,_) $\in$ activeView[$i$]
36.  if deadNodeLevel $\neq \perp$ then:
37.    activeView[$i$] $\longleftarrow$ activeView[$i$] $\setminus$ (node,_,_,_)
38.    if deadNodeLevel $<$ level then:
39.      call SelectNewParent()
40.    else if deadNodeLevel = level then:
41.      call FillActiveView()

42. Procedure SelectNewParent ():
43.  (parent, parentLvl) $\longleftarrow$ getParentInfo()
44.  lowerLevels $\longleftarrow$ {$i$ : $i <$ level $\wedge$ passiveView[$i$] $\neq \perp$ }
45.  if lowerLevels $\neq$ {} then:
46.    cLvl $\longleftarrow$ highest(lowerLevels)
47.    (candidate,_,_,_) $\longleftarrow$ getBestNode(passiveView[$cLvl$] $\cup$ activeView[$cLvl$])
48.    if candidate $\neq$ parent $\wedge$ cLvl $\geq$ parentLvl then:
49.      trigger send(Parent, $parent$, $parentLvl$, genId(self)) to $candidate$

50. Procedure FillActiveView () do:
51.  candidate $\longleftarrow$ getBestNode(passiveView[$level$])
52.  if candidate $\neq \perp$ then:
53.    if #activeView[$level$] $<$ siblingLimit then:
54.      trigger send(Hello, genId(self)) to $candidate$
55.    else if #activeView[$level$] = siblingLimit then:
56.      toRemove $\longleftarrow$ getWorstNode(activeView[$level$])
57.      if toRemove $\neq \perp \wedge$ isBetter($candidate$, $toRemove$) then:
58.        trigger send(Hello, genId(self)) to $candidate$
```

sample to return to the sender in a ShuffleReply message (Alg. 1 lines $30-31$). Upon receiving that message the sender updates the information in its active and passive views using the same strategy discussed above (Alg. 1 lines $32-33$). If no ShuffleReply message is received, the node that started this process will mark the peer used in this exchanged as being suspected of failure.

Whenever a node crashes, the tree defined by Bias Layered Tree may lose its connectivity, in which case its needs to be repaired. This is identified when a node in the active view becomes suspected. The recovery behavior depends on the level of the suspected node. If the suspected node has a level that is lower to the level of the current node (i.e., is its parent) a new parent must be chosen (Alg. 1 lines $38-39$) similarly to when the node first joined the system. If the node is in the same level, a new sibling will be chosen to replace the suspected node (Alg. 1 lines $40-41$). Finally, if the suspected node has a higher level that the local node (i.e., is a child) no local repair measure has to be taken. To handle the failure of a sibling, a node simply executes the FillActiveView procedure (Alg. 1 line 50), that uses information from the passive view to locate nodes in the same level as itself, giving preference to nodes that are closer to itself considering the IP prefix distance heuristic and sending a Hello message to that peer, enabling the peer to become aware of the existence of the local node.

4.4 Optimizing the Tree

Finally, the algorithm includes a periodical action to verify if a better parent is locally known, by executing the SelectNewParent procedure, and also selecting siblings that are closer to the local node considering the IP prefix distance heuristic, by executing the FillActiveView procedure, which are the same mechanisms introduced previously to recover from faults. Notice that when the node is optimizing the tree topology, it might need to disconnect from its current parent. Additionally, once a child node disconnects from its current parent it does not need to send an explicit message, since this disconnection will create a suspicion on the local node, leading its current parent to remove its identifier from the active view to the passive view, updating the status of the node to suspected. When picking better siblings the same can happen, in which case the node will pick its worst sibling (using the IP prefix distance heuristic) and disconnect from it.

5 Evaluation

To evaluate our solution in a cloud-edge environment, we emulated a network composed of 100 docker containers. Each container was assigned an IP address, a level, and a latency map. Nodes with small level values (i.e., closer to the cloud), were assigned more CPU quota than nodes with higher level values (i.e., closer to the edge). In fact, level zero was assigned half of the total quota, level one was assigned a third, level two a quarter, and so on. The network bandwidth available to each level was also restricted with a similar division method, starting with 1000 mbits of outgoing bandwidth to level zero. Incoming bandwidth was set as the double of outgoing, as it is common to have higher download bandwidth than upload bandwidth. At startup, each container executes a set of instructions (using the Linux `tc` tool) that apply the latency map and bandwidth restrictions appropriated for that node. The configuration of each level is summarized in Table 1.

All experiments were conducted on the Grid5000 testbed (https://www.grid5000.fr/),

Table 1: Container Parameters per Level

Level	Number of Nodes	CPU Quota	Bandwidth In/Out (Mbps)
0	1	1/2	2000 / 1000
1	3	1/3	1000 / 500
2	9	1/4	500 / 250
3	27	1/5	250 / 125
4	60	1/6	100 / 50

using 20 machines in a single cluster. Each machine has an Intel Xeon Gold 5220, with 18 cores and 96 GiB of memory. In our experiments each machine hosts 5 containers. Machines are connected through two 25 Gbps Ethernet ports to a switch.

To obtain our configuration, we first generated a random network of 10000 nodes with `inet` [29]. From this, we extracted the subgraph composed of the first 100 nodes and computed a matrix containing the shortest path distances between all nodes. As `inet` places nodes in a two dimensional plane and attributes the weights of edges (i.e., links) as the euclidean distance (between the nodes), we generated IP addresses that correlate with the distance, and multiplied the distance matrix by 0.04 effectively generating latency values between 4.76 and 831.52 milliseconds, with an average link latency of 293.39 milliseconds. We took inspiration on how BGP networks are organized [19, 24] to generate the IP addresses. To this end, we explored the graph in a breadth-first search pattern, considering each node to have as successors the 3 closest nodes (that are not already successors of any other node). For each explored node, we attributed a level value and an IP sub-network. The level value was attributed as the node's height in the search tree, while sub-networks were assigned by dividing the node's parent sub-network (e.g., if node 0 has sub-network 10.10.0.0/16, its children nodes 1, 2, and 3, have sub-networks 10.10.0.0/19, 10.10.32.0/19, and 10.10.64.0/19, respectively). To assign an IP address to a node while avoiding duplicates, we use the network IP with the last digit set to 1 plus the network prefix length (e.g., child 1 in the example above is assigned IP 10.10.0.20).

Each container executes our solution and relevant baselines as a Java application. The used baselines are: HyParView [12], X-Bot [10], Cyclon [27], and T-Man [6]. Because T-Man requires a peer sampling protocol, our implementation of T-Man takes advantage of our Cyclon implementation. X-Bot was configured to optimize the latency of overlay links, with the aid of an oracle that measures latency periodically (with UDP pings) to known peers. T-Man and Bias Layered Tree both optimize the distance considering the IP prefix distance heuristic. All protocols are implemented in the same code base and we validated each implementation independently considering the results made available in each of the papers that introduced these protocols.

Table 2 reports protocol-specific parameters used by each protocol in our experiments. The first row of the table represents the common parameters applied to all protocols. In this case, as all protocols perform a periodic shuffle operation, the period was set to 2 seconds. X-Bot additionally contains the similar parameters to HyParView, as the protocol performs optimizations over the overlay built by HyParView. Similarly, T-Man inherits Cyclon's parameters. Furthermore, all network communication performed by our implementations are via TCP connections.

As propagation of control and management information is essential to manage a complex system that spans from the cloud to the edge, we evaluate the overlay solutions when used for

Table 2: Protocol Parameters

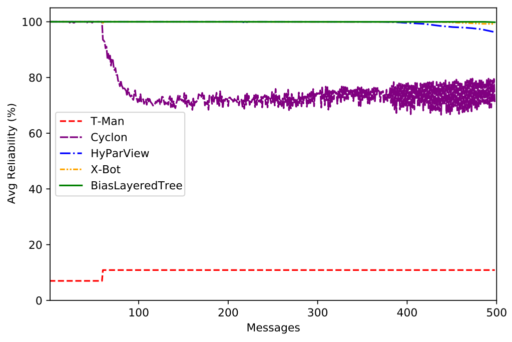
Protocol	Parameter	Value
Common	Shuffle ΔT	$2s$
Bias Layered Tree	Active View Size	3 (+1$parent$) (+$N children$)
	Passive View Size	4 (+6) (+4) (+2)
	Shuffle Sample Size	6 $(2 + 4)$
	Optimization ΔT	$2s$
	Forward Join Level Random Walk Length	3
	Levels in Forward Join	5
	Nodes per Level in Forward Join	4
	Long Distance Shuffle ΔT	$10s$
	Fill Active View ΔT	$1s$
HyParView	Active View Size	7
	Passive View Size	8
	Shuffle Sample Size	6 $(2 + 4)$
	Active Random Walk Length	2
	Passive Random Walk Length	4
X-Bot	Not-Optmized Neighbors	2
	Oracle ΔT	$2s$
	Optimization ΔT	$15s$
	Optimization Sample Size	4
Cyclon	Cache Size	15
	Shuffle Sample Size	7
T-Man	Fanout	5
	View Size	7
	Gossip ΔT	$2s$

data dissemination. We employ two different dissemination protocols. A simple, but costly, Flood Gossip protocol [12], that forwards a message to all its neighbors when it receives it for the first time; and the more efficient Plumtree [11] protocol, that embeds a tree structure on the overlay, by considering feedback from (previous) message propagations. All results reported here are an average of three independent runs. Results showed low variability across independent executions.

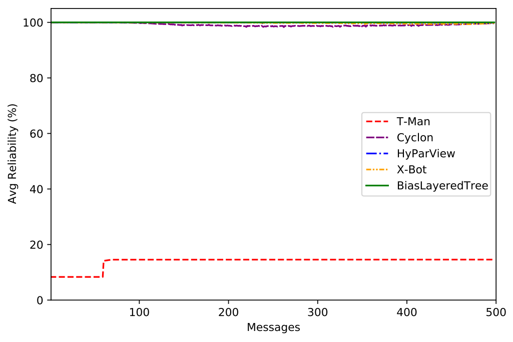
5.1 Performance Evaluation

We start by presenting the performance evaluation in fault-free scenarios. In this set of experiments, we measure and compare the average latency for delivering messages, the dissemination throughput (delivered messages per time unit), and reliability as the percentage of nodes that deliver each message. In these experiments, a 1 minute period is used for all nodes to join the overlay before beginning the dissemination of messages. No measures are taken during this period.

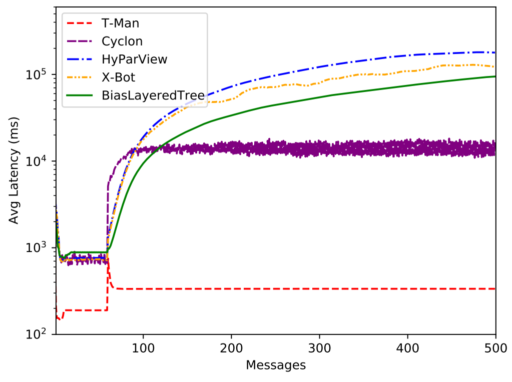
Reliability under Load Figures 1a and 1b present a measure of the average reliability (in the y axis) for each message (in the x axis) for each protocol while under heavy load using respectively, Flood and Plumtree as dissemination strategies. In this experiment, messages are generated with a frequency of 1 message per second. During the first 60 seconds, only a single node generates messages, this is required to allow Plumtree to converge to an adequate configuration. Afterwards, all nodes start to send messages for 440 seconds. Notice that this means, that after point 60 in the x axis, the average reliability represents the average reliability of 100 messages (one per node). The size of the payload of each message is 20.000 bytes, which saturates the bandwidth of the higher-level nodes. The experiment has a duration of 20 minutes to accommodate network queuing effects.



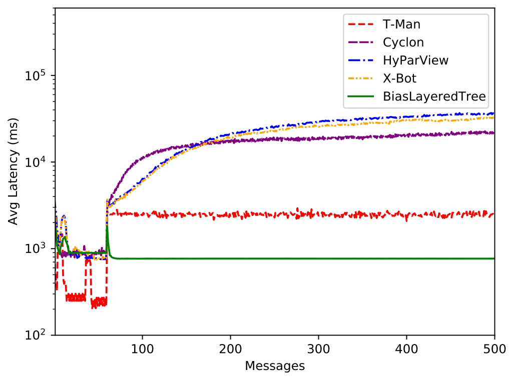
(a) Reliability using Flood



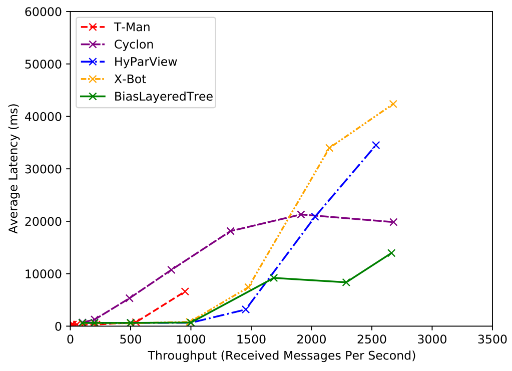
(b) Reliability using Plumtree



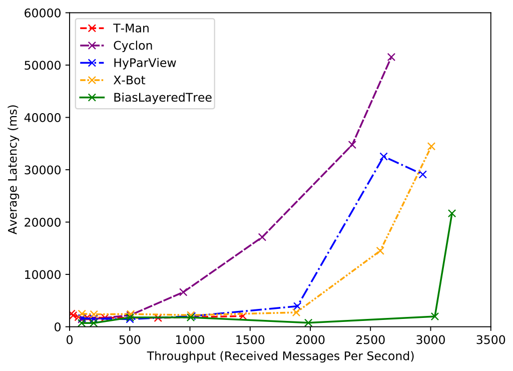
(c) Latency using Flood



(d) Latency using Plumtree



(e) Throughput vs Latency Flood



(f) Throughput vs Latency Plumtree

Figure 1: Performance of protocols under heavy load

Figure 1a shows results achieved with Flood, the results are similar for all protocols except Cyclon and T-Man. Cyclon's operation forces neighborhoods to change at every shuffle operation. This however, leads TCP connections to be closed, and consequently messages queued in the operative system (due to network saturation) to be dropped. The queuing effect is mostly visible at the end of the experiment with HyParView, where the last messages have not been delivered for all nodes. T-Man is unable to achieve a reliability of 100% as T-Man formed a network with clusters that are disconnected among them, due to its aggressive optimization strategy (this aligns with previous findings reported in [10]).

Figure 1b shows results obtained with Plumtree. We note that most protocols can maintain 100% reliability even while under heavy load. This is due to Plumtree's better usage of network resources, which is achieved by avoiding to transmit large redundant messages. The exception to this is Cyclon, since Plumtree was not designed to operate on overlays whose partial views contents change frequently. This leads Plumtree behavior to degenerate to that of a Flood protocol, yielding similar results. In both scenarios, our solution is able to achieve a broadcast reliability of 100%.

Latency under Load Figures 1c and 1d report the average latency (in the y axis in logarithmic scale) measured for each message (in the x axis) in the previous experimental setup using Flood and Plumtree respectively.

Figure 1c reports values for the Flood protocol. As expected, the latency for all solutions increases linearly as messages begin to be queued in the operating system. We notice that T-Man and Cyclon present lower latencies, but this is because their reliability is also lower (latency is only measured for messages that are delivered). For the remaining protocols, we notice that HyParView has the highest latency, this is because HyParView does not perform any optimization, which is improved by using a solution such as X-Bot.

Our solution is able to provide the lowest latency. This happens due to the way our overlay operates, by positioning nodes with higher capacity at higher points in the tree, which allows to mitigate the queuing effect due to network saturation. These effects are more prominent in experiments with Plumtree (in Figure 1d), where the latency increases for competing alternatives, while our solution maintains a stable average latency. In fact, the average latency drops after the first messages are sent by all nodes, as Plumtree is able to adjust its configuration to make a better use of available overlay links.

Throughput Figures 1e and 1f report the (maximum) throughput (in the x axis) and latency (in the y axis) observed for each protocol for increasingly rates of broadcast messages being generated by second, varying from 1 (first point on each line) to 100 messages per second per node (last point on each line). The payload size of each message is 1000 bytes. These experiments run for 4 minutes and we collect metrics of received messages and their average latency every 10 seconds. Each point represents the maximum number of messages received with minimum average latency.

Figure 1e shows the results with Flood Gossip. In this figure we can see that Cyclon begins to saturate very early, and stabilizes with an average latency of 20s, after which, as seen before, its reliability drops. We note that X-Bot is able to achieve a slightly higher throughput than HyParView albeit at the cost of higher latency. This can be explained through the fact that messages not delivered are not considered when computing the latency, which means that X-Bot is able to deliver more messages in 10 seconds but overall these messages take more time to be delivered; while HyParView is experiencing message queuing. Our solution experiences less queuing due to its hierarchical structure and therefore significantly outperforms the competing alternatives.

Figure 1f shows the results when using Plumtree to disseminate messages. In this figure we can see that, while other solutions start saturating at around 20 messages per second, our solution is able to keep an higher throughput and lower latency until approximately 50 messages per second, when it also starts saturating. As explained previously, this happens because the overlay created by our solution takes into account the bandwidth of nodes (encoded on the level), leading nodes with less bandwidth to become leafs in Plumtree dissemination tree, preventing them from becoming bottlenecks.

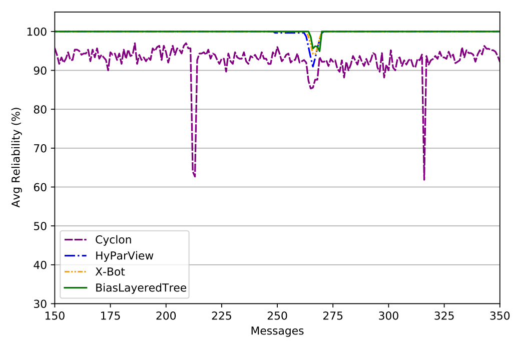
5.2 Fault-Tolerance Evaluation

Considering fault-tolerance, we tested how the various protocols react to different failure scenarios, by failing 10%, 25%, and 50% of all nodes in the network simultaneously. For these experiments, we measured reliability per message achieved by each protocol during the failure with only a single transmitter, as the goal is to evaluate the ability of the different overlays to reconfigure themselves in the presence of node failures. However, the payload of messages is increased to 500.000 bytes, to ensure that the network is close to saturation. Messages are broadcasted every second for a period of 400 seconds. Experiments have a full duration of 10 minutes with faults being induced close to the transmission of message 260 (notice however that nodes take some time before failing). As previously, we allow 1 minute for stabilization without messages being broadcasted.

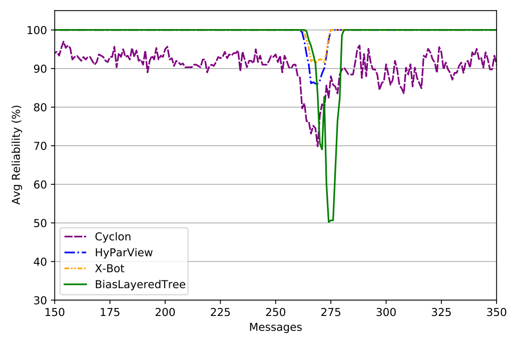
Figure 2 shows the results for the experiments with faults using Flood and Plumtree. The results show that with a low level of failures (Figs. 2a and 2d), Bias Layered Tree is able to recover faster and regain 100% reliability faster, this can be explained by the the limited impact of these faults into the tree topology. However, higher numbers of faults (statistically) affect an increasing fraction of interior nodes, leading the tree to break with a negative greater impacts in the reliability of the dissemination protocols (Figs. 2b and 2e). Finally, with 50% failures (Figs. 2c and 2f), the impact on the reliability in Bias Layered Tree is more noticeable than in competing alternatives however, the recovery speed of our tree is on par with the other protocols. This is because our recovery mechanism is based on the same principles as HyParView and X-Bot, that leverage the passive view to replace suspected nodes from the active view, albeit with additional restrictions for which we bias the contents of our passive views.

6 Conclusion

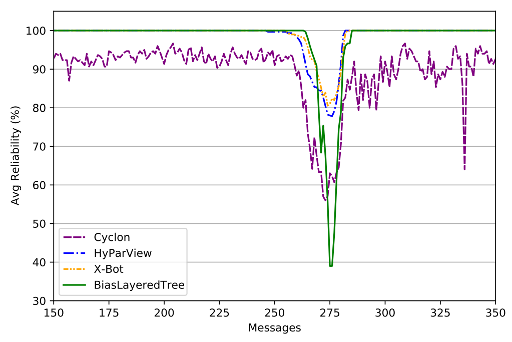
In this paper, we presented a study on the viability of adapting existing p2p solutions to manage complex systems that span from the cloud to the edge, as opposed to using small static hierarchies as the ones typically employed for Fog Computing. We also presented a novel decentralized membership protocol, named Bias Layered Tree, which takes into account the computational and network capacity available in each node, encoded in a numerical level associated with each node, and a proximity criteria based on IP prefix commonality, to build a robust hierarchical tree topology that connects and allows to manage large numbers of nodes across the cloud and edge. We evaluated our solution and relevant baselines found in the state of the art in an emulated edge network with 100 heterogeneous nodes. Our results show that it is possible to build decentralized membership management solutions that can more efficiently disseminate application and management information by adapting overlay networks to explicitly take into account capacity and distance between devices.



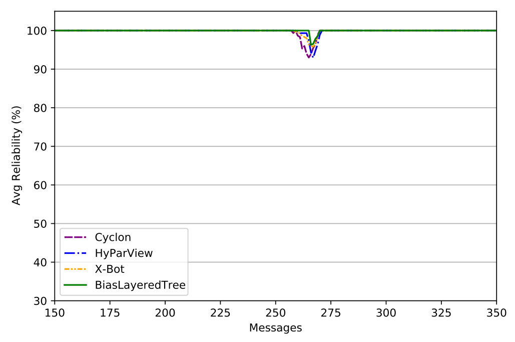
(a) Flood 10% Faults



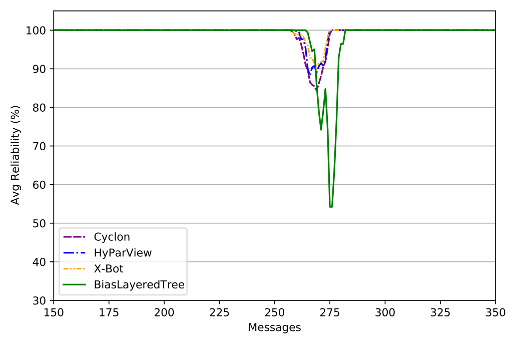
(b) Flood 25% Faults



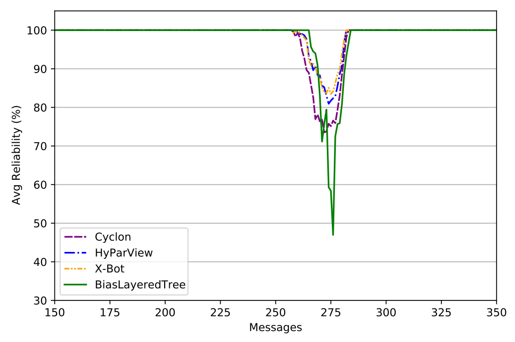
(c) Flood 50% Faults



(d) Plumtree 10% Faults



(e) Plumtree 25% Faults



(f) Plumtree 50% Faults

Figure 2: Reliability under node failures for Flood and Plumtree

References

[1] Cisco. Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are. `https://tinyurl.com/zktxmux`, 2015. Accessed: 2020-09-4.

[2] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proc. of MCC'12*, pages 13–16, Helsinki, Finland, 2012. ACM.

[3] Cisco. Cisco annual internet report (2018–2023) white paper. `https://tinyurl.com/shmk6f2`, 2020. Accessed: 2020-09-4.

[4] Pedro Ákos Costa, André Rosa, and Joã Leitão. Enabling wireless ad hoc edge systems with yggdrasil. In *Proc. of SAC'20*, CZ, 2020. ACM.

[5] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC*, volume 8, 2010.

[6] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. T-man: Gossip-based fast overlay topology construction. *Computer networks*, 53(13):2321–2339, 2009.

[7] P. Karwaczynski. Fabric: Synergistic proximity neighbour selection method. In *Proc. of P2P'07*, pages 229–230, 2007.

[8] J. Leitão. *Topology Management for Unstructured Overlay Networks*. PhD thesis, Technical University of Lisbon, 2012.

[9] J. Leitão, P. Á. Costa, M. C. Gomes, and N. Preguiça. Towards enabling novel edge-enabled applications. Technical report, 2018.

[10] J. Leitao, J. P. Marques, J. Pereira, and L. Rodrigues. X-bot: A protocol for resilient optimization of unstructured overlay networks. *IEEE TPDS*, 23(11), 11 2012.

[11] Joao Leitao, José Pereira, and Luis Rodrigues. Epidemic broadcast trees. In *Proc. of SRDS'07*. IEEE, 2007.

[12] João Leitão, José Pereira, and Luis Rodrigues. Hyparview: A membership protocol for reliable gossip-based broadcast. In *Proc. of DSN'07*. IEEE, 2007.

[13] João Leitão, Liliana Rosa, and Luis Rodrigues. Large-scale peer-to-peer autonomic monitoring. In *GLOBECOM Workshops*, pages 1–5. IEEE, November 2008.

[14] Jin Liang, Steven Y. Ko, Indranil Gupta, and Klara Nahrstedt. Mon: Management overlay networks for distributed systems. In *Proc. of SOSP'05*, page 1–2, USA, 2005. ACM.

[15] Redowan Mahmud, Ramamohanarao Kotagiri, and Rajkumar Buyya. Fog computing: A taxonomy, survey and future directions. In *Internet of Everything: Algorithms, Methodologies, Technologies and Perspectives*, pages 103–130. Springer Singapore, Singapore, 2018.

[16] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[17] Rodrigo Rodrigues and Peter Druschel. Peer-to-peer systems. *Commun. ACM*, 53(10):72–82, October 2010.

[18] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proc. of IFIP/ACM Int. Conf. on Dist. Sys. Platforms and Open Dist. Processing*, pages 329–350. Springer, 2001.

[19] Yuval Shavitt and Eran Shir. Dimes: Let the internet measure itself. *SIGCOMM Comput. Commun. Rev.*, 35(5):71–74, October 2005.

[20] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 10 2016.

[21] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Comp. Comm. Review*, 31(4), 2001.

[22] Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 189–202, 2006.

[23] William Tärneberg, Vishal Chandrasekaran, and Marty Humphrey. Experiences creating a framework for smart traffic control using aws iot. In *Proc. of UCC'16*, pages 63–69, New York, NY, USA, 2016. ACM.

[24] Y. Tian, R. Dey, Y. Liu, and K. W. Ross. Topology mapping and geolocating for china's internet. *IEEE Transactions on Parallel and Distributed Systems*, 24(9):1908–1917, 2013.

[25] Luis M. Vaquero and Luis Rodero-Merino. Finding your way in the fog: Towards a comprehensive definition of fog computing. *SIGCOMM Comput. Commun. Rev.*, 44(5):27–32, October 2014.

[26] M. Villari, M. Fazio, S. Dustdar, O. Rana, and R. Ranjan. Osmotic computing: A new paradigm for edge/cloud integration. *IEEE Cloud Computing*, 3(6):76–83, Nov 2016.

[27] Spyros Voulgaris, Daniela Gavidia, and Maarten Van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and systems Management*, 13(2):197–217, 2005.

[28] N. Wang, B. Varghese, M. Matthaiou, and D. S. Nikolopoulos. Enorm: A framework for edge node resource management. *IEEE Transactions on Services Computing*, pages 1–1, 2017.

[29] Jared Winick and Sugih Jamin. Inet-3.0: Internet topology generator. Technical report, Technical Report CSE-TR-456-02, University of Michigan, 2002.