Koala: Towards Lazy and Locality-Aware Overlays for Decentralized Clouds

Genc Tato, Marin Bertier, Cédric Tedeschi

Koala: Towards Lazy and Locality-Aware Overlays for Decentralized Clouds

Genc Tato
Univ Rennes, Inria, CNRS, IRISA
genc.tato@inria.fr

Marin Bertier
INSA Rennes, Inria, CNRS, IRISA
marin.bertier@irisa.fr

Cédric Tedeschi
Univ Rennes, Inria, CNRS, IRISA
cedric.tedeschi@inria.fr

***Abstract*—Current cloud computing infrastructures and their management are highly centralized, and therefore they suffer from limitations in terms of network latency, energy consumption, and possible legal restrictions. Decentralizing the Cloud has been recently proposed as an alternative. However, the efficient management of a geographically dispersed platform brings new challenges related to service localization, network utilization and locality-awareness. We here consider a cloud topology composed of many small datacenters geographically dispersed within the backbone network.**

In this paper, we present the design, development and experimental validation of Koala, a novel overlay network that specifically targets such a geographically distributed cloud platform. The three key characteristics of Koala are laziness, latency-awareness and topology-awareness.

By using application traffic, Koala maintains the overlay lazily while it takes locality into account in each routing decision. Although Koala's performance depends on application traffic, through simulation experiments we show that for a uniformly distributed traffic, Koala delivers similar routing complexity and reduced latency compared to a traditional proactive protocol, such as Chord. Additionally, we show that despite its passive maintenance, Koala can appropriately deal with churn by keeping the number of routing failures low, without significantly degrading the routing performance. Finally, we show how such an overlay adapts to a decentralized cloud composed of multiple small datacenters.

I. Introduction

Over the last decade, as a result of both industrial and academic research, cloud computing has entirely reshaped the way computing infrastructures are managed. Major industry actors such as Amazon and Microsoft are currently the main owners of the cloud infrastructure. Recently, these companies have built very large datacenters in various locations around the globe, trying to move computation closer to their clients. Still, their architecture remains primarily centralized. Centralized architectures have certain advantages, such as easy management and more optimization opportunities. Nevertheless, the distance between a centralized datacenter and its clients is often significant. This results in increased latency, higher energy consumption [3], and is subject to legal restrictions [20].

Various initiatives, such as the Discovery project [1], have suggested an alternative to such a centralized architecture by bringing the Cloud even closer to its clients. They propose a fully decentralized architecture which leverages the existing Internet Service Provider (ISP) backbone network infrastructures, by introducing computing resources in Point of Presence (PoP) within the backbone. In this vision, each PoP potentially hosts a small datacenter which interacts with other small datacenters (in other PoPs) in a decentralized, peer-to-peer (P2P), way in order to provide the same services as the traditional Clouds, but at a reduced cost.

A first challenge brought by decentralization is the needed functionality of service localization. In a centralized architecture, there is a single point of reference able to provide information about the status of services/resources available in the platform. In a decentralized Cloud instead, no single component has a global knowledge of the system, yet every node in the system should be able to process any query, based on a partial view of the platform. In other words, we need an overlay network to define these partial views and a routing mechanism which allows us to efficiently reach the server in charge of a specific service given a service ID.

A second challenge is the efficient network utilization. In centralized Clouds, system maintenance traffic stays generally within the datacenter and uses dedicated links so that it does not interfere with user traffic. In decentralized Clouds instead, maintenance traffic, same as user traffic, is distributed, and they both share the same communication links.

For this reason, minimizing the management traffic is essential in order to save bandwidth for the users.

A third challenge is the reduction of latency by leveraging locality, one of the main motivations of decentralized Clouds. We distinguish two kinds of locality-awareness, namely *inter-PoP* and *intra-PoP*. Inter-PoP locality-awareness means that the logical path of a message travelling between different PoPs should resemble as much as possible to the physical one. We refer to this kind of locality-awareness as *latency-awareness* hereafter.

Intra-PoP locality-awareness means that a message between nodes in the same PoP should not leave this PoP. We refer to this as *topology-awareness* hereafter.

In order to address these challenges we have recently described Koala [25], a structured overlay network for efficient service localization which minimizes management traffic while providing locality-aware routing. While in [25] we present the key design features of Koala, the current paper presents Koala's architecture and algorithms and their validation through a comprehensive set of simulations.

Koala nodes use the Kleinberg model [8] based on logical distance (difference in node identifiers) for filling their routing

tables. While Kleinberg defines the ideal entries for a node's routing table, Koala nodes do not actively search for nodes matching these ideal entries, but rather fill the routing tables opportunistically, when such nodes come to their knowledge through information piggybacked in the application traffic. In other words, if no traffic is generated by the application, there is no traffic at all within the overlay. Koala's performance depends on the amount of traffic generated by applications running on top of it (for instance a cloud stack). Regarding locality-awareness, Koala focuses primarily in reducing inter-PoP latencies by making nodes select each routing hop according to a tradeoff between a latency-based and a greedy routing based on logical distance. Koala also provides a way to take the multi-PoP topology into account through integrating the information of which datacenter a node belongs to in its identifier, thus enabling intra-PoP routing when possible.

Our experiments show that despite its lazy maintenance, Koala still delivers a similar complexity to proactive protocols, while it further reduces latencies due to its latency-aware routing. Additionally, they show that passive maintenance is sufficient for repairing the overlay in case of reasonable churn. Finally, we show that Koala adapts well to a multi-datacenter environment.

The remainder of this paper is organized as follows. Section II presents Koala's related work. Section III presents the details of the protocol. Section IV presents our simulation results. Section V concludes.

II. Related Work

A. Structured, unstructured or hybrid?

Overlay networks are generally either *structured* or *unstructured*. Structured overlays offer efficient data retrieval mechanisms but they rely on the maintenance of a specific routing structure which can become costly to maintain as churn increases. On the other hand, unstructured overlays are designed to be resilient to churn, but this benefit comes at the cost of redundant periodic messages, and hence a very high network utilization. Additionally, unstructured overlays are not designed to support efficient unicast. Aiming at combining the best of both worlds, hybrid overlays can be found in literature. These are generally structured overlays built on top of unstructured ones. In this case the unstructured overlay is responsible for the membership management and the structured one for the efficient routing.

Structured overlays are commonly used to build distributed hash tables (DHTs). DHTs such as Chord [24], Pastry [22] or Kademlia [16] focus on guaranteeing message delivery in a logarithmic number of hops. This relies on strict rules for filling routing tables. Maintaining such a rigid structure is costly, therefore other protocols introduced some flexibility into the routing table construction. The authors of Symphony [14] show that the same complexity can be achieved by choosing the *long links* randomly, yet carefully (based on a specific probability distribution). However, in all these cases there is a background mechanism for detecting and repairing inconsistencies in the routing table. These mechanisms run either continuously or periodically, and therefore they have a high cost on network utilization.

Unstructured overlays, on the other hand, do not build their routing tables (or partial views) based on strict rules. For instance, HyParView [12] selects neighbors entirely randomly, while Scamp [6] and CLON [15] use a probabilistic model for accepting new neighbors based on the number a node has currently. This simplifies the maintenance of the partial view in case of failures because nodes do not need to search for specific entries. However, for the overlay to be efficient, nodes need to continuously exchange these partial views (using gossiping). This can be done periodically, as in HyParView, or reactively, as in Scamp. Reactive gossiping happens only when nodes detect some change. CLON is an adaptation of Scamp for multi-clouds. It improves Scamp's partial view exchange by favoring nodes in the same datacenter more than those in different ones. Regarding the network utilization, unstructured overlays are expensive not only when adopting a periodic approach but also when adopting a reactive one. This is because for the overlay to be reactive, connections need to be continuously monitored.

In order to benefit from the reliability of unstructured overlays and the efficiency of unicast in structured ones, several studies focus on hybrid overlays. T-Man [7] and Fluidify [21] are examples where structured overlays are created as a result of filtering information which comes from gossiping. T-Man provides a generic interface which can be used to create structured overlays based on custom criteria. Fluidify instead, is more specific and it uses gossiping to select nodes which are physically close to improve locality-awareness. Even though these protocols combine the advantages of both types of overlays, they still suffer from the same drawbacks when it comes to network utilization.

B. The scale of laziness

As a structured overlay, Koala shares many core aspects with similar overlays, such as Chord and Symphony. However, it does not have any background mechanism for detecting and repairing the routing table. Problems are detected and repaired in a lazy way, only when routing application traffic by means of piggybacking. Piggybacking is used also in Kademlia, but only partially, to improve multiple routings between the same source and destination. The concept of using application traffic for building overlays is used also in [9]. This study assumes that the application issues repeatedly similar queries, and therefore it links together nodes which collaborated in previous queries. This means that the overlay is adapted to a specific application. Koala instead, uses application traffic to optimize the overlay independently of the applications running on top of it. Relax-DHT [11] follows a lazy approach on data replication, by allowing replicas to get further from the root as the network evolves. In this way it avoids the need for a systematic rearrangement of data blocks whenever a node joins or leaves. However, this technique reduces data maintenance rather than overlay maintenance as in Koala. A concept of laziness similar to the one of Koala is introduced in [19],

but in an entirely different context, namely the locality-aware placement of virtual machines used by a given application. Figure 1 shows an overview of the overlay families we discussed above ordered by their laziness.

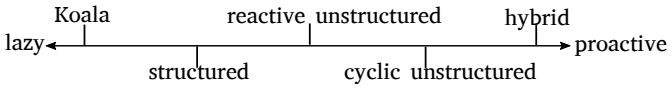

Fig. 1: Ordering overlays according to laziness.

In this paper we do not quantitatively compare the laziness of Koala with respect to other proactive overlays as this depends on certain parameters which can vary, such as the periodicity for gossiping or for running the maintenance protocol. We rather insist on the conceptual difference and show that despite being fully lazy, Koala can perform at least as good or even better than non-lazy protocols. For our comparison, we have used Chord as a representative of structured overlays which has been shown to offer the best trade-off between bandwidth used and lookup latency [13].

C. Locality-awareness

Latency-awareness can be introduced into overlays through different mechanisms, namely (i) topology-aware overlay construction, (ii) topology-aware neighbor selection, and (iii) proximity routing [2]. *Topology-aware overlay construction* consists in choosing logical neighbours according to their physical proximity. Fluidify [21] follows this path: it updates the logical identifiers according to the identifiers of physically close nodes. A similar technique is used by Vivaldi [4], but for placing nodes in a Cartesian space according to latencies experienced between nodes. In contrast to Fluidify, which receives latencies from an underlying gossip protocol, Vivaldi uses application traffic, and therefore is lazier.

Topology-aware neighbor selection consists in choosing the physically closest node among viable candidates. This is done only when there is some flexibility to fill a given entry of the routing table. This is why T-Chord [17] relaxes Chord's constraints for having a link at a precise distance, by allowing a range of values around this distance. Links with the lowest latency satisfying the logical constraints are selected for each entry. However, similar to Fluidify, T-Chord is based on the same gossip underlying mechanism. In Pastry [22] instead, nodes obtain lazily low-latency neighbors during the joining phase but then need to actively update them at a second stage.

Proximity routing consists in taking routing decisions not only according to the remaining logical distance to the destination, but also on the delay of the next hop. It is the lightest approach, since it does not require any routing table modification. This approach has been adopted by Hypeer [23], which aims at introducing latency-awareness in Chord by changing the order of its hops. Its authors show that by making Chord choose the next hop based also on latency improves its overall performance. Koala also uses primarily this technique due to its lightness. Still, Koala is more flexible than Hypeer as it is not restricted by the order on hops.

III. Protocol

A. Model

We consider a distributed system consisting of nodes which communicate via message passing. These nodes are grouped into different physical locations called Points of Presence (PoPs) and are interconnected so that a message sent from one node can be delivered, in finite time, to any other one. We assume that the latency of a message within the same PoP is insignificant compared to the one between different PoPs. We assume that a communication channel between any two given nodes can be established and that these channels are fair-lossy, meaning that the channels can lose a finite number of messages. Nodes can join or leave (crash) at any time. The crash can be permanent or the node can recover, but then it has to re-join the system. As we aim to use Koala to interconnect nodes in different PoPs, we assume a reasonable level of churn.

B. Koala: under the hood

Similar to Chord, nodes in Koala are organized logically in a ring by assigning them a unique m-bit ID from a circular ID space, as shown in Figure 2. However, in contrast to Chord, routing in Koala is bidirectional, meaning that the logical distance d between two IDs is the smallest between the distances calculated in both directions, clockwise and counterclockwise, and it is defined as follows:

$$d(id_1, id_2) = min(|id_1 - id_2|, 2^m - |id_1 - id_2|) \quad (1)$$

Each node has a routing table. For each entry of the routing table, a node stores four fields: i) the logical ID, ii) the IP, iii) the latency when reaching that entry expressed in Round Trip Time (RTT) and iv) an Ideal ID (IID) field that we explain below. Routing entries can be classified in two main groups: *core links* and *optimization links*.

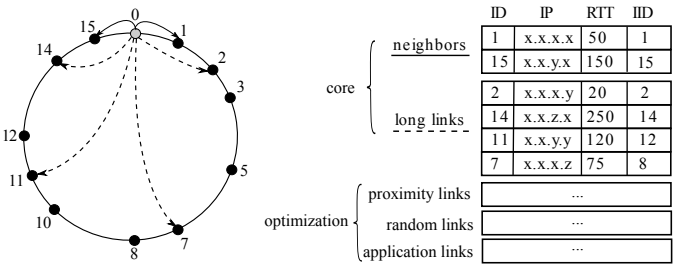

Fig. 2: Logical ring in a small network (m=4), and the routing table of node 0.

1) ***Core links****:* are entries which are essential for enabling efficient routing. They can be either *neighbors* or *long links*.

Neighbors are nodes with predecessor or successor IDs in the identifier space. They ensure the ring structure is maintained so that every node is reachable. A node can keep multiple pairs of neighbors, for instance, 2 predecessors and 2 successors.

Long links are entries which enable shortcuts in the ring structure and affect routing significantly. In order to provide a logarithmic routing, long links need to be carefully selected

at different logical distances from the node itself. In Koala we do this by using a continuous version of Kleinberg distribution which is defined as follows: $p_n(d) = 1/(d\ln(M))$. This formula defines the probability to have a long link at a distance d, where M is the maximum distance in the ring. Given that Koala is bidirectional, this distance is half of the ring, 2^{m-1}. Note that this distribution favors shorter distances over long ones, enabling logarithmic routing. As we detail below, we use this probability distribution to generate the ideal distances at which our long links should be. Consecutively, we convert these distances into IDs by randomly either adding or subtracting them from the ID of the node itself. The generated IDs constitute the ideal IDs (IIDs) for our entries. Note that generating the IID fields for long links does not mean that nodes with this IID exist. We could actively search for them, but that is against the principle of laziness. Therefore, we rather wait to learn about them (or nodes with close IDs to them) in a process we describe later.

*2) **Optimization links**:* are additional links which are not necessary for the functioning of the overlay but that can optimize it further. We focus particularly on *proximity links*, but other kinds of links, such as *random links* or *application links* can be used as well in certain situations. *Proximity links* are a distinct set of nodes with which a node has communicated in the past and which have a small RTT value. They are ordered in a fixed size queue based on the RTT value. As entries with smaller latency are added, the ones with higher ones are removed. The combination of long links, which are selected based on their logical distance, and proximity links, which are based on their latency, provides a good input for our latency-based routing algorithm we explain shortly.

Random links and *application links* can be used to improve the diversity of entries for nodes with similar IDs and to adapt the overlay to the application, respectively. Random links are particularly useful when nodes in the same PoP share a part of their ID. Similarly, application links are handy when considering particular application patterns. In this paper we consider a uniformly distributed traffic, and therefore impact of these links is left as a future work.

The number of entries in each group is configurable. For Kleinberg distribution to provide $O(\log N)$ routing (N being the size of the network), we need at least m long links. Therefore, for long links we choose a multiple of m such as $Nr_{ll} = C_{ll} * m$. We configure the number of long links by setting the value of C_{ll}. Although not necessary for the routing complexity, we can do the same thing with the size of the other groups of links so that it remains relative to the size of the system. Thus, C_n, C_{pl} can control the number of neighbors and proximity links, respectively.

C. Routing: greedy, latency-based, or both?

Once we have defined the routing table, we need to define a routing algorithm which determines the next hop for a message. The Kleinberg distribution of long links provides us with a logarithmic complexity in terms of number of hops if a greedy algorithm is used. Greedy routing selects the entry which reduces the most the remaining logical distance to the destination. Nevertheless, it does not consider the cost of each hop in terms of latency. As a result, even though the number of hops is reduced, the cost of the total path might still high. On the other hand, an algorithm which selects the entry with the lowest latency, without considering the logical distance, does not provide correctness and can increase drastically the number of hops. We examine an algorithm which takes into account both factors, logical distance and latency and provides a tradeoff between them. This is done by rating each potential entry as follows:

$$R_{entry} = 1/(\alpha * d(entry.ID, dest.ID) + (1-\alpha) * norm(entry.RTT)) \quad (2)$$

where $d(entry.ID, dest.ID)$ is the remaining logical distance if we choose the entry, $norm()$ is a normalization function which converts the value of RTT into the same scale as the logical distance, and α is a coefficient which determines the weight of each of the factors. An $\alpha = 1$ results in a purely greedy algorithm, and an $\alpha = 0$ in a purely latency-based one. Note that the normalization function maps the minimum and maximum values for the remaining distance to those of the RTT, and derives the values in between using a linear function. The remaining distance can vary from 0, when the next step is the final destination, to $d(node.ID, dest.ID) - 1$ if we progress just by one step. For the RTT, the maximum value can be introduced empirically (if the estimation is smaller than the real value, the consequence is the same as decreasing α. Similarly, a higher estimation is equivalent to increasing α.

Algorithm 1 Routing.

```
1: function onRoute(msg)
2:   if msg.dest ≠ this then
3:     if msg.src = null then
4:       initializePiggyBack(msg)                ▷ 4 - 5: explained in Section III-E
5:     msg.piggybackRT.add(this.asEntry())
6:     fwd ← this.getNextHop(msg.dest)
7:     if fwd ∈ this.RT.neighbors.succs then     ▷ 7 - 10: explained in Section III-F
8:       msg.piggybackRT.add(this.RT.neighbors.preds)
9:     if fwd ∈ this.RT.neighbors.preds then
10:      msg.piggybackRT.add(this.RT.neighbors.succs)
11:    send(fwd, msg)
12:  else
13:    notifyApplication(msg)
14: function getNextHop(dest)
15:   maxRating ← -1, maxRatingEntry ← null
16:   for each entry ∈ this.RT do
17:     rating ← getRating(entry, dest)
18:     if rating > maxRating then
19:       maxRating ← rating, maxRatingEntry ← entry
20:   return maxRatingEntry
21: function getRating(entry, dest)
22:   if d(this.ID, dest.ID) < d(entry.ID, dest.ID) then
23:     return -1
24:   if d(entry.ID, dest.ID) = 0 then
25:     return ∞
26:   return 1 / ( α * d(entry.ID, dest.ID) + (1 - α) * norm(entry.RTT))
```

The details of the routing algorithm is provided by Algoritm 1. When a node receives an application message the *OnRoute* methods is called. If this node is the destination, the message is sent to the application, otherwise it is forwarded to the node with the highest rating in the routing table (in *getNextHop*). Note that in the function *getRating* we discard

nodes for which the distance to the destination is smaller than the one of the node itself (Line 23). This provides correctness even when $\alpha = 0$. Additionally, if the destination is present in the routing table, we forward directly to that entry without taking into account the latency (Line 25).

D. Joining

As shown in Algorithm 2, before joining the network a node generates its own unique m-bit ID by hashing its IP using a arbitrary-length output hash function (Line 2). Based on this ID it initializes its long links. As we explained before, for each link we generate random distances according to Kleinberg distribution and convert them into IIDs by randomly adding or substracting them from the ID of the node. Note that initially, the ID, IP and RTT fields are set to some default values (Line 9). ID_{df} is a special value such that its distance from any other ID is infinity: $d(ID_{df}, *) = \infty$. IP_{df} is an empty string, whereas RTT_{df} is set to an approximate value of the average RTT in the system. After the initialization phase, the node tries to join the network by sending a "exchange routing table" request to an existing node (commonly called the *bootstrap* node) which IP address is known beforehand (Line 4).

Algorithm 2 Joining.

```
1: function onJoin( )
2:     this.ID = Hash(this.IP, m)
3:     this.intializeLongLinks()
4:     send(this.bootNode, ExhangeRTMsg(this.RT + [this.asEntry()]))
5: function intializeLongLinks( )
6:     this.RT.longlinks ← [ ]
7:     for i = 1 → C_ll * m do
8:         newIID ← this.getKleinbergIID()
9:         newLL ← Entry(ID_df,IP_df, newIID, RTT_df)
10:        this.RT.longlinks.add(newLL)
11: function getKleinbergIID( )
12:    d ← round(2^(RandomDouble(0,1)*(m−1)))
13:    if RandomInt() % 2 = 0 then
14:        return (this.ID+d) %2^m
15:    return (this.ID-d+2^m) %2^m
16: function onExchangeRT(msg)
17:    oldNeighbors ← this.getImmediateNeighbors()
18:    for each entry ∈ msg.srcRT do ▷ srcRT contains parameters of ExhangeRTMsg
19:        updateLinks(entry, this.RT.longlinks)          ▷ omitting proximity links
20:        updateNeighbors(entry)
21:    newNeighbors ← this.getImmediateNeighbors()
22:    for each entry ∈ newNeighbors do
23:        if entry ∉ oldNeighbors then
24:            send(entry, ExhangeRTMsg(this.RT + [this.asEntry()] + oldNeighbors))
25:    if msg.src ∉ newNeighbors then
26:        fwd ← this.getNextHop(msg.src)
27:        send(fwd, msg)
28: function getImmediateNeighbors( )
29:    return [this.RT.neighbors.succs[0], this.RT.neighbors.preds[0]]
30: function updateLinks(newEntry, rt)
31:    for each entry ∈ rt do
32:        if d(entry.ID, entry.IID) > d(newEntry.ID, entry.IID) then
33:            entry.ID ← newEntry.ID
34:            entry.IP ← newEntry.IP
35:            entry.RTT ← newEntry.RTT
```

This request triggers the *onExchangeRT* handler on the recipient node. The latter examines the received routing table, which includes all parameters passed to the *ExhangeRTMsg* constructor, and checks if there are neighbors or links of interest. That is, a link which ID is closer to the IID of any of the current links than the ID of the link itself. If such a link is found we update our current links as shown in method *updateLinks*. Since the sender just joined, the received table consists of only the sender. Assuming that the first recipient is not an immediate neighbor of the joining node, the request is forwarded to an entry in the routing table (Line 27). Forwarding will continue until a neighbor of the joining node is reached.

The neighbor will reply to the joining node by sending its routing table, its own entry details and its old neighbors (Line 24). The joining node, would look through the received entries and update its long links (generally copy them) and also find the other immediate neighbor, which was the old neighbor of the sender. Consequently, the joining node would contact it as well and receive from it its routing table which will be used to update its long links. Note that from the pseudo-code above an extra message is sent to the first discovered neighbor but that can be avoided by specifying a flag for the exchange request which distinguishes joining requests from the others. As a result of these exchanges, the joining node will find its correct position in the ring and will have a list of links by merging the routing tables of its immediate neighbors.

E. Lazy learning

As described above, a node learns from its neighbor about links of interest already during joining. However, learning does not stop there, but it continues permanently as messages are routed. Each routed message is provided with a data structure very similar to the routing table of the nodes (the *msg.piggybackRT*). At each step during routing, this structure is enriched with information about existing nodes. One source of this information is the path of the message. Whenever a node forwards a message, it piggybacks on it its own details as shown in Line 5 of Algorithm 1.

A second source of piggybacked information is the routing tables of nodes in the message's path. The source node which starts the routing request initializes also the piggyback structure of the message as shown in Algorithm 3. This is very similar to the initialization of the long links except that the distribution is rather uniform. This is done to improve the diversity of the information coming from the first source as this does not depend strictly on the path itself. Note that we keep the size of the piggyback structure logarithmic to the size of the network as we do with all the other structures using the C_{pb} setting. The value of C_{pb} is chosen based on a tradeoff between fast learning and message overhead.

Algorithm 3 Learning while routing.

```
1: function intializePiggyBack(msg)
2:     msg.piggybackRT ← [ ]
3:     for i = 1 → C_pb * m do
4:         newIID ← RandomInt(2^m)
5:         newP ← Entry(ID_df,IP_df, newIID, RTT_df)
6:         msg.piggybackRT.add(newP)
7: function onReceiveMsg(msg)
8:     for each piggyEntry ∈ msg.piggybackRT do
9:         updateLinks(piggyEntry, this.RT.longlinks)     ▷ omitting proximity links
10:        updateNeighbors(piggyEntry)
11:    for each entry ∈ this.RT do
12:        updateLinks(entry, msg.piggybackRT)
```

Whenever a message is received the *onReceiveMsg* handler is called. Initially the receiver node will look in the piggybacked structure for links of interest and neighbors as done during routing table exchanges. At a second step, it will try to update this structure with links from its own routing table. Therefore, at each step a node receives information about the nodes present in the network and distributes also its own knowledge. These simple mechanism can be thought of as passive gossiping. Each node gossips information about the state of the network, but without periodically contacting any node solely for this purpose. As a result, the speed of learning about the network depends directly on the amount of application traffic.

F. Resilience

The extent to which a protocol is resilient to changes in the network depends on its ability to maintain the links in the routing table. However, periodic maintenance procedures are not compatible with our main principle, which is being as lazy as possible. For this reason, our protocol is not designed to support high degrees of churn. Nevertheless, we take a few inexpensive measures to support churn to some degree.

Node joining and departing are handled differently depending on whether they are neighbors or long links. In case they are neighbors, taking action immediately is much more important than when they are long links. As described before, when a node joins the network, its future neighbors are immediately notified. On the contrary, upon a node departure, its neighbors are not notified until one of them tries to contact it. At this point, it will remove it from the routing table and will exchange routing tables with the next node in its neighbors list so they become immediate neighbors. For this to work, the list of neighbors needs to be up to date. We do this by piggybacking neighbor information when messages are exchanged between neighbors. For instance, when node q sends a message to its successor s, it also reports its predecessor p. So in case q departs, s and p will exchange routing tables. This is shown in lines 7 - 10 of Algorithm 1. This lazy mechanism for knowing neighbors' neighbors does not guarantee that the information will not be outdated. In case a node loses all its neighbors in a certain direction (for example, all the predecessors), it will try to search for its new neighbors by contacting a working long link in the vicinity (possibly in the direction of the lost neighbors). This is a special search because it targets nodes between the long link and the node, therefore it follows always the same direction in the ring. If no intermediate nodes are found, then the long link becomes the new neighbor.

In case a long link joins or departs, the management is more straightforward. Joining nodes which might be ideal long links for current nodes are noticed only as information gets disseminated using piggybacking. A node will realize that a long link is down only when trying to forward a message to it. In that case, it will forward the message to the link with the second highest rating according to Equation 2, and it will mark this link as down. That means that this link can be replaced with other links, even with a higher difference form the IID. In order to avoid re-adding links, we timestamp each link using a Lamport clocks [10]. When the rating of all entries in the routing table are smaller than the one of the node itself, the node declares failure to the sender of the message. This happens only in cases of extreme churn, when all the nodes in the routing table with a closer ID to the destination are down, including neighbors, which can be temporarily unavailable during the neighbor searching phase.

IV. Experimental Validation

In order to evaluate our overlay, we have conducted various experiments using the PeerSim [18] simulator. For each experiment, we use the same basic setup. We assign random coordinates to each PoP on a unit 2D-coordinate system. Nodes within the same PoP have the same coordinates. We use the Waxman model [26] for creating links between these PoPs. In this model the coefficients α and β are selected in such a way that the model respects roughly the neighbor degree of Renater[1], a research ISP network in France. This model does not guarantee a connected graph at once. Therefore we recursively apply Waxman on randomly picked nodes from the connected sub-graphs until the graph is connected. The resulting topology represents our *physical network*. In order to simulate the IP routing in the physical network, we use Dijkstra shortest path algorithm [5], where the cost of an edge is a function of the Euclidian distance between the nodes. The cost of a physical path represents its RTT. Consequently, the RTT of a logical path in Koala is the sum of RTTs of all physical paths composing it.

In order to compare Koala with physical routing or other protocols, the simulator assigns the exact same list of tasks to all protocols. A task can be: routing a message, adding a node, or removing a node. At the end of a task, metrics such as RTT, physical and logical hops, and failures are collected and compared. Although we use PeerSim in the event-based mode, tasks are processed on a cyclic basis. At each cycle one or more tasks can be processed. Our reported results are based on the average values of a group of cycles. We show the size of this group in round brackets in the X-axis label.

We conduct four sets of experiments, presented in sections IV-A to IV-D, which focus on: Koala's scalability, latency-awareness, reliability and topology-awareness, respectively. The first three sets concentrate on Koala's inter-PoP aspects, and therefore we assume each node belongs to a different PoP, *i.e.* we have one node per PoP. However, in the fourth set, we focus on topology-awareness, and therefore we consider multiple nodes per PoP.

A. Number of long links and scalability

In the first experiment, we study the behavior of our overlay as the network scales up. Following a uniform traffic distribution (we plan to work with real traces in our future work), at each cycle, one node joins the network and also a

[1]https://www.renater.fr

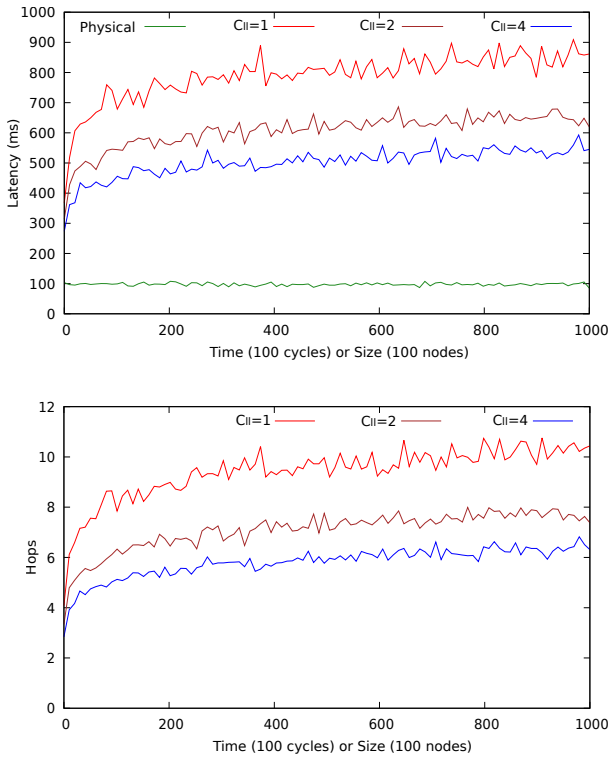

Fig. 3: Latency and hops as network scales up to 100K nodes.

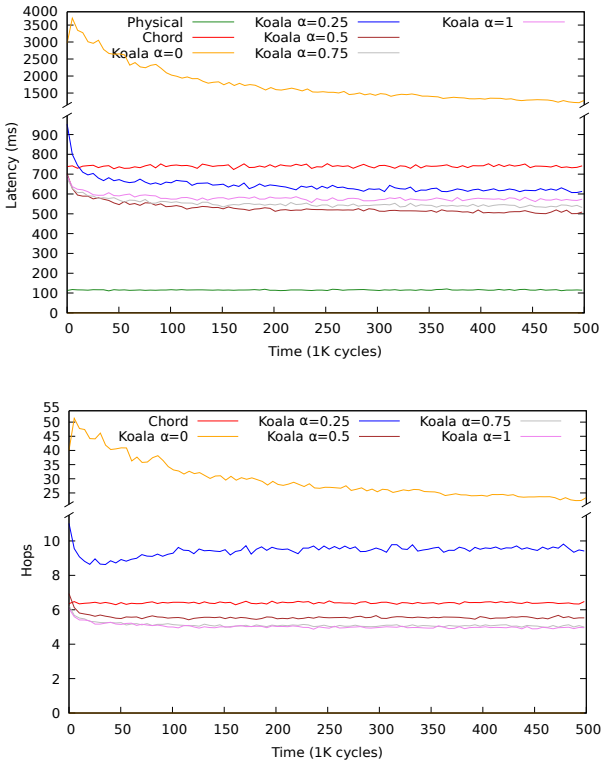

Fig. 4: Impact of α on latency and hops.

message is sent from a random source to a random destination. We scale up to 100K nodes and assume no node leaves the network. We run the same experiment for different numbers of long links, by varying the constant C_{ll}. We fix α to 1, therefore routing does not take the RTT into account. As for the rest of the experiments, the number of neighbor pairs is set to 2 ($C_n = 2/m$, 2 successors, 2 predecessors), and the piggyback list size to m ($C_{pb} = 1$). In this case proximity links are not considered ($C_{pl} = 0$). Figure 3 shows how latency and logical hops are affected as the network scales up (physical hops are omitted as we focus on the comparison of logical ones).

As mentioned in section III-B, $Nr_{ll} = C_{ll} * m$. For $N = 100K$, $m = 17$, therefore $C_{ll} = 1, 2, 4$ correspond to $17, 34$ and 68 long links. Note that one point in the graph represents the average of 100 routes. In Figure 3, we observe that regardless of the number of links, both latency and hops grow logarithmically as more nodes join. It is clear that the higher the number of long links, the lower the latency and the hops. However, by doubling the number of links, we do not reduce twice the latency. Therefore, at some point, increasing further the number of long links would not provide significant benefit, while it might introduce additional latency in case of multiple attempts to connect to stale long links.

B. Lazy learning and latency-awareness

In the following experiment, we demonstrate the ability of Koala to discover the network in a lazy way, as application messages are routed. We particularly focus on the impact of the parameter α on latency and hops. We use a static network of 10K nodes, where no nodes join or leave the network. For this experiment we compare our protocol with Chord. At each cycle, both protocols are requested to route a message from the same random source to the same random destination. As in the previous experiment, we fix the number of long links by setting $C_{ll} = 2$ (28 long links) and we do not consider proximity links. For Chord we use 4 successors (equal to Koala's neighbors). We repeat this experiment by varying α from 0 to 1, by adding each time 0.25. Figure 4 shows the impact of this variation on latency and number of hops.

We observe that regardless of the value of α, as more messages are routed, Koala nodes discover long links which are closer to their ideal ones, thus latency and hops decrease over time. Note that this learning is faster at the beginning. As the discovered long links comply more and more with the Kleinberg model, finding the exact ideal link does not result in significant latency improvements.

Concerning the impact of α on latency and hops, we notice that for $\alpha = 0$, when the routing decisions are almost not at all based on logical distance, Koala delivers a poor performance. This is because at each hop a message is always forwarded to nearby nodes without significantly approaching the destination, which leads to many hops and as a consequence, in very high latencies. However, if we consider slightly more the logical distance ($\alpha = 0.25$), we still have a relatively high number of hops, but we significantly improve the latency

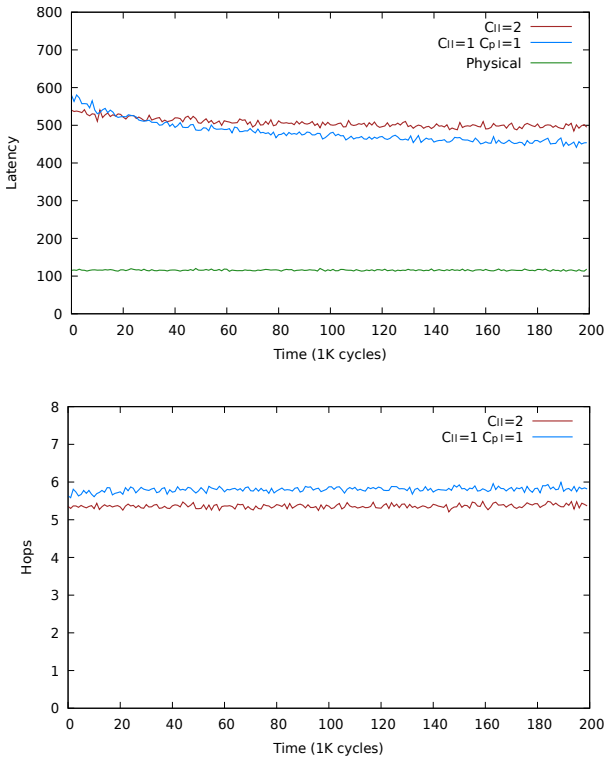

Fig. 5: Impact of proximity links on latency and hops.

compared to when $\alpha = 0$. On the other hand, for $\alpha = 1$, when routing is based solely on logical distance, we achieve the lowest number of hops, but that does not mean the lowest latency. To improve latency, one needs to take RTT more into account by reducing α. This can be done until an equilibrium is found where reducing it more results in higher latency as the number of hops also increases. For our simulation, this equilibrium happened to be for $\alpha = 0.5$, but this value depends on various factors such as network topology and traffic pattern.

In addition, we notice that for this setting Koala delivers the messages up to 32% faster than Chord. This is not only because Koala uses more long links (accountable for 65% of the gain when $\alpha = 1$), but also due to its latency-awareness (accountable for the additional 35% of the gain when $\alpha = 0.5$).

So far we have considered only links selected on their distance criteria, and therefore the tradeoff routing function is mainly determined by the logical distance. We continue the previous experiment by considering proximity links too. We do not change the overall number of links, instead we divide them equally between long links and proximity links. Therefore, we compare using $C_{ll} = 2$ with $C_{ll} = 1$ and $C_{pl} = 1$. We do this for the best setting from the previous experiment, when $\alpha = 0.5$ and fix this setting for the rest of the experiments.

In Figure 5 we observe that when combining long links and proximity links, the number of hops is slightly higher than when using only long links. However, as more and more proximity links are found the routes are more latency-aware and the delivery time is improved (by additional 10%).

C. Resilience to churn

Differently from proactive protocols which focus on avoiding failures by actively maintaining their routing table, Koala focuses on repairing itself once the failure has already happened. Therefore, in the next two experiments we study Koala's ability to repair itself under churn.

In the first experiment we consider a network of 5K nodes. As before, at each cycle a message is routed between a random source and a random destination. However, every 80 cycles, CHURN nodes leave the network, and the same number join. These two events do not necessarily happen in the same cycle. In all the experiments in this section we use $C_{ll} = 2$ and no proximity links as we mainly focus on keeping the Kleinberg properties under churn. We analyze how Koala deals with various levels of churn by varying the number CHURN.

Figure 6 shows the effect of churn on latency (upper part) and failure rate, *i.e.*, percentage of messages that failed to reach their destination (lower part). In the upper part of this figure we notice that, as the level of churn increases, the routing latency of successfully delivered messages increases as well, but rather gracefully. This is due to our learning techniques which help nodes to update their stale links without significantly breaking the Kleinberg distribution. Nevertheless, the same thing cannot be said for the number of failures. The lower part of Figure 6 shows that for low levels of churn (CHURN $= 1$ or 2), the lazy repairing is efficient enough to keep failure rate low. However, as churn levels increase (CHURN $= 8$), the overlay does not process enough traffic per unit of time in order to repair before other changes in the network occur. In that case the failure rate continuously grows, but note that this is an extreme level of churn.

In the next experiment we analyze the ability of Koala to repair itself in case of an abrupt failure of a whole section of the network (CHURN_SEC). In this case, we consider a network of 10K nodes with no joins. At each cycle, we route as in the previous experiments, but at cycle 5K we introduce the unexpected failure. We vary the size of the failed section and show its effects on latency and failure rate in Figure 7.

As the amount of long links per node affected by this sort of failure is significantly higher than in the previous experiment, the Kleinberg distribution is initially broken, and this results in higher latencies. However, as the ring starts repairing itself and the number of nodes is reduced, the latencies are reduced as well. Nevertheless, the lower part of Figure 7 shows that the ability of Koala to repair depends on the amount of failed nodes. We can observe that the higher this amount, the longer it takes for the overlay to rebuild itself. Moreover, if this section of the network is too large (90%), Koala is not able to fully recover anymore, but this is rather unrealistic.

D. Topology-awareness

Until now we have considered only one node per PoP in order to focus on inter-PoP aspects of Koala. We now consider a topology of PoPs having multiple nodes. We examine two

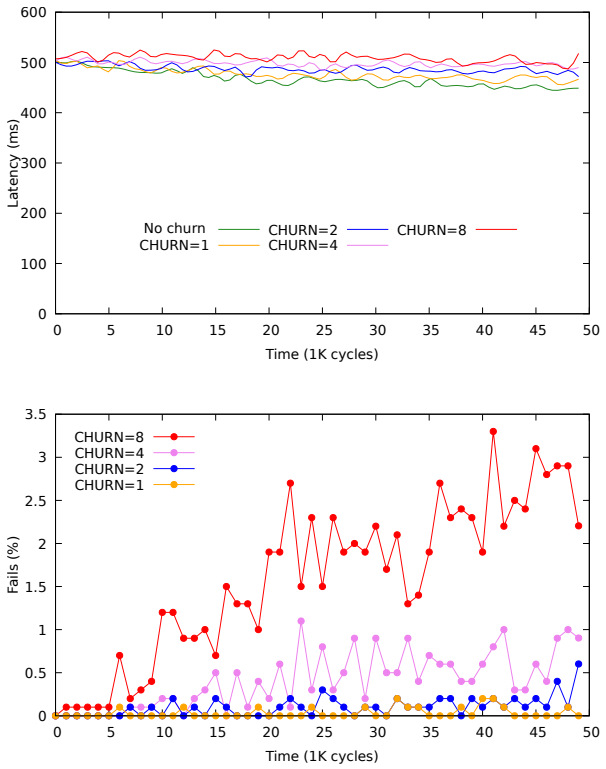

Fig. 6: Impact of churn on latency and failure rate.

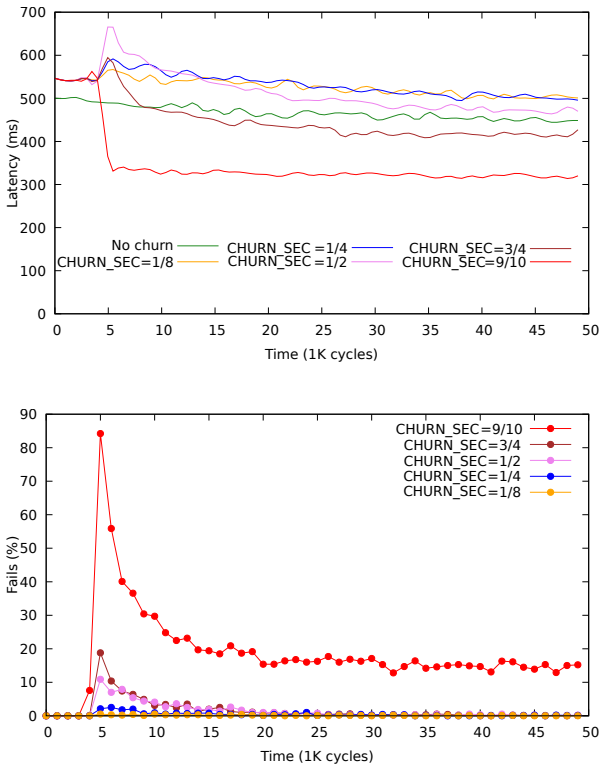

Fig. 7: Impact of a massive failure on latency and failure rate.

cases: when Koala is unaware of which nodes belong to which PoPs, and when Koala is given this information. In the first case, node IDs are generated entirely randomly, and thus nodes belonging to the same PoP might have distant IDs. In the second case instead, we assume we know whether nodes belong to the same PoP, and therefore we assign them consecutive IDs. This is done by reserving a range of IDs for each PoP, but it requires a rough estimation of the maximum number of PoPs and of nodes per PoP.

For this experiment we consider a network of 10K nodes divided in 1K PoPs (10 nodes per PoP). As in experiment IV-B, the network is static, and at each cycle a message is sent from a random source to a random destination. We use the same number of long links and proximity ones ($C_{ll} = 1$, $C_{pl} = 1$). We run this experiment twice, once when node IDs are assigned randomly, and once when nodes in the same PoP are assigned consecutive IDs, for both Koala and Chord.

In Figure 8 we observe that when IDs are fully random we obtain very similar results as in experiment IV-B, where we considered one node per PoP. This is because, with a high probability, nodes in the same PoP will not be in each others' routing table. Therefore, even when routing within the same PoP, the message might travel through different other PoPs before returning to the initial one. Consequently, nodes behave as if they were in different PoPs and they do not take advantage of the low intra-PoP latencies.

On the other hand, when nodes in the same PoP are assigned consecutive IDs, we notice that both Koala and Chord improve their latency. However, Koala delivers a greater improvement ($\approx 28\%$) than Chord ($\approx 16\%$). This difference is significant given that the lower the latencies of the messages, the more difficult it is to further improve them, and Koala already delivered better latencies than Chord. The reason why Koala exploits better this extra information is the routing tradeoff function we explained in section III-C. In Koala, a non-local message will initially travel locally within the PoP (using very cheap hops and small gains in logical distance) until it finds a good option, for which jumping outside the PoP is more convenient as the gain in logical distance justifies the higher latency. In this way, Koala makes better decisions before choosing the next PoP. On the contrary, Chord would choose to send a non-local message immediately to another PoP by trying to reduce the distance in IDs without consulting before local nodes. However, Chord's hops become smaller as the message approaches the destination, and thus Chord's final hops are also within the same PoP. Therefore, Chord also exploits the PoP topology, but to limited extent.

V. Conclusions and Future Directions

Koala is an overlay network designed to target decentralized cloud environments. It provides efficient, locality-aware service localization while it minimizes traffic by eliminating useless overlay maintenance. When the application stops communicating, Koala stops communicating too. Our experiments

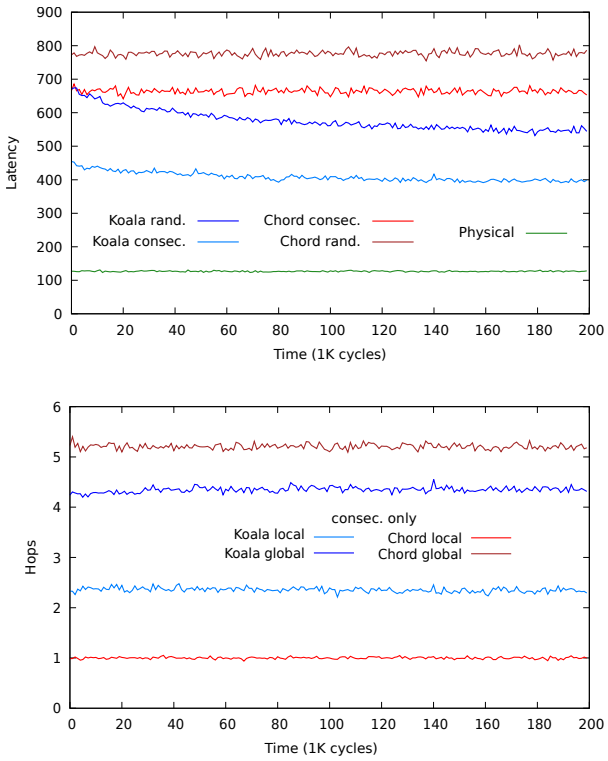

Fig. 8: Impact of cloud topology on Koala and Chord.

show that, despite its laziness, Koala still guarantees a similar performance to other traditional protocols by providing a $O(\log(N))$ complexity. They suggest that by finely tuning the degree of locality-awareness, we can reduce further the inter-PoP latencies. Additionally, experiments demonstrate that Koala can appropriately handle churn by lazily repairing the overlay. Finally, we show that just by assigning consecutive identifiers to nodes in the same PoP, Koala becomes intra-PoP locality-aware as well and it adapts to the decentralized cloud topology much better than Chord. In this paper we have focused on the core features of Koala which deal mainly with inter-PoP aspects. Our next step is to formalize the intra-PoP locality-awareness without the need to reserve identifier intervals for the different PoPs. We plan to split the node identifier in two parts: one part for the PoP, and another part identifying the node within the PoP, enabling a distinction between intra and inter-PoP routing policies and thus in a reduction of the ring diameter. Additionally, we intend to investigate intra-PoP collaboration for a better inter-PoP routing without introducing hierarchies. On more practical aspects, we plan to devise the integration of Koala into a decentralized OpenStack, to support decentralized messaging, as conveyed by the Discovery project [1].

References

[1] M. Bertier, F. Desprez, G. Fedak, A. Lebre, A.-C. Orgerie, J. Pastor, F. Quesnel, J. Rouzaud-Cornabas, and C. Tedeschi, *Cloud Computing: Challenges, Limitations and R&D Solutions*, 2014, ch. Beyond the Clouds: How Should Next Generation Utility Computing Infrastructures Be Designed?

[2] M. Castro, P. Druschel, Y. C. Hu, and A. I. T. Rowstron, "Topology-Aware Routing in Structured Peer-to-Peer Overlay Networks," in *Future Directions in Distributed Computing, Research and Position Papers*. Springer, 2003.

[3] I. Cuadrado Cordero, A.-C. Orgerie, and C. A. Morin, "GRaNADA: A Network-Aware and Energy-Efficient PaaS Cloud Architecture," in *IEEE International Conference on Green Computing and Communications (GreenCom)*, Sydney, Australia, Dec. 2015.

[4] F. Dabek, R. Cox, F. Kaashoek, and R. Morris, "Vivaldi: A decentralized network coordinate system," *SIGCOMM Comput. Commun. Rev.*, 2004.

[5] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, no. 1, pp. 269–271, Dec. 1959. [Online]. Available: http://dx.doi.org/10.1007/BF01386390

[6] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié, *Third International COST264 Workshop*, 2001, ch. Scamp: Peer-to-Peer Lightweight Membership Service for Large-Scale Group Communication.

[7] M. Jelasity and O. Babaoglu, *Third International Workshop on Engineering Self-Organising Systems (ESOA 2005)*. Springer, 2006, ch. T-Man: Gossip-Based Overlay Topology Management, pp. 1–15.

[8] J. Kleinberg, "The small-world phenomenon: An algorithmic perspective," in *Proceedings of ACM STOC'00*. ACM, 2000, pp. 163–170.

[9] G. Koloniari, Y. Petrakis, E. Pitoura, and T. Tsotsos, "Query workload-aware overlay construction using histograms," in *Proceedings of the 14th ACM International Conference on Information and Knowledge Management*, ser. CIKM '05. ACM, 2005, pp. 640–647.

[10] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[11] S. Legtchenko, S. Monnet, P. Sens, and G. Muller, "RelaxDHT: A Churn-Resilient Replication Strategy for Peer-to-Peer Distributed Hash Tables," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 7, no. 2, pp. 28:1–28:18, 2012.

[12] J. Leitao, J. Pereira, and L. Rodrigues, "Hyparview: A membership protocol for reliable gossip-based broadcast," in *International Conference on Dependable Systems and Networks (DSN'07)*, 2007.

[13] J. Li, J. Stribling, T. M. Gil, R. Morris, and M. F. Kaashoek, *Comparing the Performance of Distributed Hash Tables Under Churn*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 87–99.

[14] G. S. Manku, M. Bawa, and P. Raghavan, "Symphony: Distributed Hashing in a Small World," in *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, Seattle, USA, Mar. 2003.

[15] M. Matos, A. Sousa, J. Pereira, R. Oliveira, E. Deliot, and P. Murray, *CLON: Overlay Networks and Gossip Protocols for Cloud Environments*. Springer, 2009.

[16] P. Maymounkov and D. Mazières, "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric," in *First International Workshop on Peer-to-Peer Systems (IPTPS'02)*. Springer, 2002, pp. 53–65.

[17] A. Montresor, M. Jelasity, and O. Babaoglu, "Chord on demand," in *Fifth IEEE International Conference on Peer-to-Peer Computing*, 2005.

[18] A. Montresor and M. Jelasity, "PeerSim: A scalable P2P simulator," in *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, Sep. 2009.

[19] J. Pastor, M. Bertier, F. Desprez, A. Lebre, F. Quesnel, and C. Tedeschi, "Locality-aware cooperation for VM scheduling in distributed clouds," in *Proceedings of Euro-Par 2014*, pp. 330–341.

[20] S. Pearson and A. Benameur, "Privacy, security and trust issues arising from cloud computing," in *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, Nov 2010, pp. 693–702.

[21] A. C. Resmi and F. Taiani, *Fluidify: Decentralized Overlay Deployment in a Multi-cloud World*. Cham: Springer International Publishing, 2015.

[22] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems," in *Proceedings of IFIP/ACM Middleware'01*, 2001, pp. 329–350.

[23] S. Serbu, P. Felber, and P. Kropf, "HyPeer: Structured Overlay with Flexible-Choice Routing," *Computer Networks*, no. 1, 2011.

[24] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications," in *Proceedings of ACM SIGCOMM'01*, 2001, pp. 149–160.

[25] G. Tato, M. Bertier, and C. Tedeschi, "Designing Overlay Networks for Decentralized Clouds," in *International Workshop on the Future of Cloud Computing and Cloud Services (FutureCloud 2017)*, Hong-Kong, Dec. 2017. [Online]. Available: https://hal.inria.fr/hal-01634409

[26] B. M. Waxman, "Routing of multipoint connections," *IEEE Journal on Selected Areas in Communications*, vol. 6, no. 9, 1988.