Peer-to-Peer Netw Appl
DOI 10.1007/s12083-009-0037-7

Dealing with network partitions in structured overlay networks

Tallat M. Shafaat · Ali Ghodsi · Seif Haridi

Received: 21 October 2007 / Accepted: 17 February 2009

Abstract Structured overlay networks form a major class of peer-to-peer systems, which are touted for their abilities to scale, tolerate failures, and self-manage. Any long-lived Internet-scale distributed system is destined to face network partitions. Although the problem of network partitions and mergers is highly related to fault-tolerance and self-management in large-scale systems, it has hardly been studied in the context of structured peer-to-peer systems. These systems have mainly been studied under churn (frequent joins/failures), which as a side effect solves the problem of network partitions, as it is similar to massive node failures. Yet, the crucial aspect of network mergers has been ignored. In fact, it has been claimed that ring-based structured overlay networks, which constitute the majority of the structured overlays, are intrinsically ill-suited for merging rings. In this paper, we present an algorithm for merging multiple similar ring-based overlays when the underlying network merges. We examine the solution in dynamic conditions, showing how our solution is resilient to churn during the merger, something widely believed to be difficult or impossible. We evaluate the algorithm for various scenarios and show that even when falsely detecting a merger, the algorithm quickly terminates and does not clutter the network with many messages. The algorithm is flexible as the tradeoff between message complexity and time complexity can be adjusted by a parameter.

Keywords DHTs · Network partitions · Network mergers · Structured overlay networks · Loopy rings · Distributed hash tables

1 Introduction

Structured Overlay Networks (SONs)—such as Chord [39], Pastry [34], and SkipNet [17]—are touted for their ability to provide scalability, fault-tolerance, and self-management, making them well-suited for Internet-scale distributed applications. Such Internet-scale systems will always come across network partitions, especially if the system is long-lived. Although the problem of network partitions and mergers is highly related to fault-tolerance and self-management in large-scale systems, it has, with few exceptions, been ignored in the context of structured overlays. This is peculiar, as the importance of the problem has long been known in other problem domains, such as in distributed databases [10] and distributed file systems [40].

Although network partitions are not very common, they do occur. Internet failures, resulting in partitioned networks can occur due to large area link failure, router failure, physical damage to a link/router, router misconfiguration and buggy software updates. Overloaded

T. M. Shafaat (✉) · S. Haridi
KTH - Royal Institute of Technology,
Electrum 229, 164 40 Kista, Sweden
e-mail: tallat@kth.se

S. Haridi
e-mail: haridi@kth.se

A. Ghodsi
Swedish Institute of Computer Science (SICS),
Box 1263, 164 29 Kista, Sweden
e-mail: ali@sics.se

Published online: 14 March 2009

routers, network wide congestion due to denial of service (DoS) attacks and routing loops [32] can also have the same effect as a network partition. Similarly, natural disasters can result in Internet failures. This was observed when an earthquake in Taiwan in December 2006 exposed the issue that global traffic passes through a small number of seismically active "choke points" [30]. Several countries in the region connect to the outside world through these choke points. A number of similar events leading to Internet failures have occurred [6]. On a smaller scale, the aforementioned causes can disconnect an entire organization from the Internet [31], thus partitioning the organization.

It is our firm belief that a crucial requirement for practical SONs is that they should be able to deal with network partitions and mergers. As we show in Section 2, SONs can, by a coincidence, cope with network partitions. Unfortunately, most SONs cannot cope with network mergers.

In fact, it has been claimed that ring-based structured overlays, which constitute the absolute majority of the SONs, are inherently a poor fit for dealing with network mergers. Datta et al. [8] focus on the merging of multiple SONs after a network partition ceases (network merger). They argue that ring-based SONs "cannot function at all until the whole merge process is complete".

The contribution of this paper is an algorithm for merging any number of similar structured overlays. We will limit ourselves to unidirectional ring-based overlays, since they constitute a majority of the SONs. The presented algorithm allows the system designer to adjust, through a *fanout* parameter, the tradeoff between bandwidth consumption (message and bit complexity) and time it takes for the algorithm to complete (time complexity). Through experimental evaluation, we show typical fanout values for which our algorithm completes quickly, while keeping the bandwidth consumption at an acceptable level. We examine the solution in dynamic conditions, showing how our solution is resilient to churn during the merger, something believed to be impossible [8]. We verify that the algorithm works efficiently even if only a single node detects the partition merger. We show that even with large rings with thousands of nodes, our solution is lean as it avoids positive-feedback cycles and, hence, avoids congesting the network. Finally, we show that the algorithm can recover from pathological scenarios, such as loopy rings [25, 38], which might result from network partitions.

The merging of SONs gives rise to problems on two different levels: *routing level* and *data level*. The routing level is concerned with healing of the routing information after a partition merger.

The data level is concerned with the consistency of the data items stored in the SONs. The solutions to this problem might depend on the application and on the semantics of the data operations, e.g. immutable key/value pairs or monotonically increasing values. It is also known that it is impossible to achieve strong (atomic) data consistency, availability,[1] and partition tolerance in SONs [5, 14, 15].

We focus on the problem of dealing with partition mergers at the routing level. Given a solution to the problem at the routing level, it is generally known how to achieve weaker types of data consistency, such as eventual consistency [11, 40].

Outline Section 2 serves as a background by motivating and defining our choice of ring-based SONs. Section 3 introduces the main contributions of this work, *simple ring unification algorithm*, as well as the *gossip-based ring unification algorithm*. Since the latter algorithm builds on the previous, we hope that this has a didactic value. Thereafter, Section 4 evaluates different aspects of the algorithms in various scenarios. Section 5 presents related work. Finally, Section 6 concludes and presents an ambitious agenda for future work.

2 Background

The rest of the paper focuses on ring-based structured overlay networks. Next, we motivate this choice, and thereafter briefly define ring-based SONs. Finally, we show how Chord deals with network partitions and failures.

Motivation for the unidirectional ring geometry We confine ourselves to unidirectional ring-based SONs, such as Chord [39], SkipNet [17], DKS [14], Koorde [20], Mercury [4], Symphony [28], EpiChord [22], and Accordion [23]. But we believe that our algorithms can be adapted easily to other ring-based SONs, such as Pastry [34]. For a more detailed account on directionality and structure in SONs, please refer to Onana et al. [3] and Aberer et al. [1].

The reason for confining ourselves to ring-based SONs is twofold. First, ring-based SONs constitute a majority of the SONs. Second, Gummadi et al. [16] diligently compared the geometries of different SONs,

[1] By availability we mean that a get/put operation should eventually complete.

and showed that the ring geometry is the one most resilient to failures, while it is just as good as the other geometries when it comes to proximity.

To simplify the presentation of our algorithms, we use notation that indicates the use of the Chord [39] SON. But the ideas are directly applicable to all unidirectional ring-based SONs.

A model of a ring-based SON A SON makes use of an *identifier space*, which for our purposes is defined as a set of integers $\{0, 1, \cdots, \mathcal{N}-1\}$, where $\mathcal{N}$ is some apriori fixed, large, and globally known integer. This identifier space is perceived as a ring that wraps around at $\mathcal{N}-1$.

Every node in the system, has a unique identifier from the identifier space. Node identifiers are typically assumed to be uniformly distributed on the identifier space. Each node keeps a pointer, *succ*, to its *successor* on the ring. The successor of a node with identifier p is the first node found going in clockwise direction on the ring starting at p. Similarly, every node also has a pointer, *pred*, to its *predecessor* on the ring. The predecessor of a node with identifier q is the first node met going in anti-clockwise direction on the ring starting at q. A *successor-list* is also maintained at every node r, which consists of r's c immediate successors, where c is typically set to $\log_2(n)$, where n is the network size.

Ring-based SONs also maintain additional routing pointers on top of the ring to enhance routing. To be concrete, assume that these are placed as in Chord. Hence, each node p keeps a pointer to the successor of the identifier $p + 2^i \pmod{\mathcal{N}}$ for $0 \leq i < \log_2(\mathcal{N})$. Our results can easily be adapted to other schemes for placing these additional pointers.

Dealing with partitions and failures in chord Chord handles joins and leaves using a protocol called *periodic stabilization*. Leaves are handled by having each node periodically check whether *pred* is alive, and setting $pred := nil$ if it is found dead. Moreover, each node periodically checks to see if *succ* is alive. If it is found to be dead, it is replaced by the closest alive successor in the successor-list.

Joins are also handled periodically. A joining node makes a lookup to find its successor s on the ring, and sets $succ := s$. Each node periodically asks for its successor's *pred* pointer, and updates *succ* if it finds a closer successor. Thereafter, the node notifies its current *succ* about its own existence, such that the successor can update its *pred* pointer if it finds that the notifying node is a closer predecessor than *pred*. Hence, any joining node is eventually properly incorporated into the ring.

As we mentioned previously, a single node cannot distinguish massive simultaneous node failures from a network partition. As periodic stabilization can handle massive failures [25], it also recovers from network partitions, making each component of the partition eventually form its own ring. We have simulated such scenarios and confirmed these results. The problem that remains unsolved, which is the focus of the rest of the paper, is how several independent rings can efficiently be merged.

3 Ring merging

For two or more rings to be merged, at least one node needs to have knowledge about at least one node in another ring. This is facilitated by the use of *passive lists*. Whenever a node detects that another node has failed, it puts the failed node, with its routing information[2] in its passive list. Every node periodically pings nodes in its passive list to detect if a failed node is alive again. When this occurs, it starts a ring merging algorithm. Hence, a network partition will result in many nodes being placed in passive lists. When the underlying network merges, this will be detected and rectified through the execution of a ring merging algorithm.

A ring merging algorithm can also be invoked in other ways than described above. For example, it could occur that two SONs are created independently of each other, but later their administrators decide to merge them due to overlapping interests. It could also be that a network partition has lasted so long, that all nodes in the rings have been replaced, making the contents of the passive lists useless. In cases such as these, a system administrator can manually insert an alive node from another ring into the passive list of any of the nodes. The ring merger algorithm will take care of the rest.

The detection of an alive node in a passive list does not necessarily indicate the merger of a partition. It might be the case that a single node is incorrectly detected as failed due to a premature timeout of a failure detector. The ring merging algorithm should be able to cope with this by trying to ensure that such false-positives will terminate the algorithm quickly. It might also be the case that a previously failed node rejoins the network, or that a node with the same overlay and network address as a previously failed node joins the ring. Such cases are dealt with by associating with every

[2]By routing information we mean a node's overlay identifier, network address, and nonce value (explained shortly).

node a globally unique random *nonce*, which is generated each time a node joins the network. Hence, if the algorithm detects that a node in its passive list is again alive, it can compare the node's current nonce value with that in the passive list to avoid a false-positive, as that node is likely a different node that coincidentally has the same overlay and network address.

3.1 Simple ring unification

In this section, we present the simple ring unification algorithm (Algorithm 1). As we later show, the algorithm will merge the rings in $O(N)$ time for a network size of N. Later, we show how the algorithm can be improved to make it complete the merger in substantially less time.

Algorithm 1 Simple Ring Unification Algorithm

```
1: every γ time units and detqueue ≠ ∅ at p
2:     q := detqueue.dequeue()
3:     sendto p : MLOOKUP (q)
4:     sendto q : MLOOKUP (p)
5: end event

6: receipt of MLOOKUP (id) from m at n
7:     if id ≠ n and id ≠ succ then
8:         if id ∈ (n, succ) then
9:             sendto id : TRYMERGE (n, succ)
10:        else if id ∈ (pred, n) then
11:            sendto id : TRYMERGE (pred, n)
12:        else
13:            sendto closestprecedingnode(id) : MLOOKUP (id)
14:        end if
15:    end if
16: end event

17: receipt of TRYMERGE (cpred, csucc) from m at n
18:    sendto n : MLOOKUP (csucc)
19:    if csucc ∈ (n, succ) then
20:        succ := csucc
21:    end if
22:    sendto n : MLOOKUP (cpred)
23:    if cpred ∈ (pred, n) then
24:        pred := cpred
25:    end if
26: end event
```

Algorithm 1 makes use of a queue called *detqueue*, which will contain any alive nodes found in the passive list. The queue is periodically checked by every node p, and if it is non-empty, the first node q in the list is picked to start a ring merger. Ideally, p and q will be on two different rings. But even so, the distance between p and q on the identifier space might be very large, as the passive list can contain any previously failed node. Hence, the event MLOOKUP(id) is used to get closer to id through a lookup. Once MLOOKUP(id) gets near its destination id, it triggers the event TRYMERGE(a, b), which tries to do the actual merging by updating *pred* and *succ* pointers to a and b respectively.

The event MLOOKUP(id) is similar to a Chord lookup, which tries to do a greedy search towards the destination id. One difference is that it terminates the lookup if it reaches the destination and locally finds that it cannot merge the rings. More precisely, this happens if MLOOKUP(id) is executed at id itself, or at a node whose successor is id. If an MLOOKUP(id) executed at n finds that id is between n and n's successor, it terminates the MLOOKUP and starts merging the rings by calling TRYMERGE. Another difference between MLOOKUP and an ordinary Chord lookup is that an MLOOKUP(id) executed at n also terminates and starts merging the rings if it finds that id is between n's predecessor and n. Thus, the merge will proceed in both clockwise and anti-clockwise direction.

The event TRYMERGE takes as parameters a candidate predecessor, *cpred*, and a candidate successor *csucc*, and attempts to update the current node's *pred* and *succ* pointers. It also makes two recursive calls to MLOOKUP, one towards *cpred*, and one towards *csucc*. This recursive call attempts to continue the merging in both directions. Figure 1 shows the working of the algorithm.

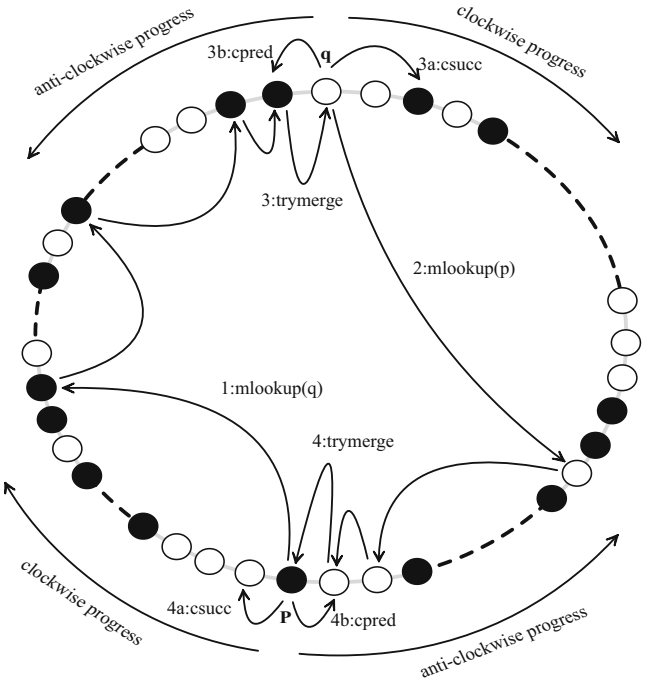

Fig. 1 *Filled circles* belong to SON1 and *empty circles* belong to SON2. The algorithm starts when p detects q, p makes an MLOOKUP to q and asks q to make an MLOOKUP to p

In summary, MLOOKUP closes in on the target area where a potential merger can happen, and TRYMERGE attempts to do local merging and advancing the merge process in both directions by triggering new MLOOKUPs.

3.2 Gossip-based ring unification

The simple ring unification presented in the previous section has two disadvantages. First, it is slow, as it takes $O(N)$ time to complete the ring unification. Second, it cannot recover from certain pathological scenarios. For example, assume two distinct rings in which every node points to its successor and predecessor in its own ring. Assume furthermore that the additional pointers of every node point to nodes in the other ring. In such a case, an *mlookup* will immediately leave the initiating node's ring, and hence may terminate. We do not see how such a pathological scenario could occur due to a partition, but the *gossip-based ring unification algorithm* (Algorithm 2) rectifies both disadvantages of the simple ring unification algorithm. Moreover, the simple ring unification is less robust to churn, as we discuss in the evaluation section.

Algorithm 2 Gossip-based Ring Unification Algorithm

```
1: every γ time units and detqueue ≠ ∅ at p
2:     ⟨q, f⟩ := detqueue.dequeue()
3:     sendto p : MLOOKUP (q, f)
4:     sendto q : MLOOKUP (p, f)
5: end event

6: receipt of MLOOKUP (id, f) from m at n
7:     if id ≠ n and id ≠ succ then
8:         if f > 1 then
9:             f := f − 1
10:            r := randomnodeinRT()
11:            at r : detqueue.enqueue(⟨id, f⟩)
12:        end if
13:        if id ∈ (n, succ) then
14:            sendto id : TRYMERGE (n, succ)
15:        else if id ∈ (pred, n) then
16:            sendto id : TRYMERGE (pred, n)
17:        else
18:            sendto closestprecedingnode(id) : MLOOKUP (id, f)
19:        end if
20:    end if
21: end event

22: receipt of TRYMERGE (cpred, csucc) from m at n
23:    sendto n : MLOOKUP (csucc, F)
24:    if csucc ∈ (n, succ) then
25:        succ := csucc
26:    end if
27:    sendto n : MLOOKUP (cpred, F)
28:    if cpred ∈ (pred, n) then
29:        pred := cpred
30:    end if
31: end event
```

Algorithm 2 is, as its name suggests, gossip-based. The algorithm is essentially the same as the simple ring unification algorithm, with a few additions. The intuition is to have the initiator of the algorithm to immediately start multiple instances of the simple algorithm at random nodes, with uniform distribution. But since the initiator's pointers are not uniformly distributed, the process of picking random nodes is incorporated into MLOOKUP. Thus, MLOOKUP(*id*) is augmented so that the current node randomly picks a node *r* in its current routing table and starts a ring merger between *id* and *r*. This change alone would, however, consume too much resources.

Two mechanisms are employed to prevent the algorithm from consuming too many messages, which could give rise to positive feedback cycles that congest the network. First, instead of immediately triggering an MLOOKUP at a random node, the event is placed in the corresponding node's *detqueue*, which is only checked periodically. Second, a constant number of random MLOOKUPs are created. This is regulated by a fanout parameter called *F*. Thus, the fanout is decreased each time a random node is picked, and the random process is only started if the fanout is larger than 1. The *detqueue*, therefore, holds tuples, which contain a node identifier and the current fanout parameter. Similarly, MLOOKUP takes the current fanout as a parameter. The rate for periodically checking the *detqueue* can be adjusted to control the rate at which the algorithm generates messages.

4 Evaluation

In this section, we evaluate the two algorithms from various aspects and in different scenarios. There are two measures of interest: *message complexity*, and *time complexity*. We differentiate between the *completion* and *termination* of the algorithm. By completion we mean the time when the rings have merged. By termination we mean the time when the algorithm terminates sending any more messages. Unless said otherwise, message complexity is until termination, while time complexity is until completion.

The evaluations are done in a stochastic discrete event simulator [37] in which we implemented Chord. The simulator uses an exponential distribution for the inter-arrival time between events (joins and failures). To make the simulations scale, the simulator is not packet-level. The time to send a message is an exponentially distributed random variable. The values in the graphs indicate averages of 20 runs with different random seeds.

We first evaluate the message and time complexity of the algorithms in a typical scenario where after merger, many nodes simultaneously detect alive nodes in their passive lists. Next, we evaluate the performance of the algorithm for a worst case scenario when only a single node detects the existence of another ring. The worst case scenario is similar to a case where an administrator wants to merge two SONs and triggers the ring unification algorithm on only a single node. Next, we assess the algorithms for a loopy ring. Thereafter, we evaluate the performance of the algorithms while node joins and failures are taking place during the ring merging process. Next, we compare our algorithm with a self-stabilizing algorithm. Finally, we evaluate the message complexity of the algorithms when a node falsely believes that it has detected another ring.

For the first experiment, the simulation scenario had the following structure. Initially nodes join and fail. After a certain number of nodes are part of the system, we insert a partition event, upon which the simulator divides the set of nodes into as many components as requested by the partition event, dividing the nodes randomly into the partitions but maintaining an approximate ratio specified. For our simulations, we create two partitions. A partition event is implemented using lottery scheduling [42] to define the size of each partition. The simulator then drops all messages sent from nodes in one partition to nodes in another partition, thus simulating a network partition in the underlying network and therefore triggering the failure handling algorithms (see Sections 2 and 3). Furthermore, node join and fail events are triggered in each partitioned component. Thereafter, a network merger event simply allows messages to reach other network components, triggering the detection of alive nodes in the passive lists, and hence starting the ring unification algorithms.

We simulated the simple ring unification algorithm and the gossip-based ring unification algorithm for partitions creating two components, and for fanout values from 1 to 7. For all the simulation graphs to follow, a fanout of 1 represents the simple ring unification algorithm. A time unit was equal to the time it takes for a message to reach its destination node.

Figures 2 and 3 show the time and message complexity for a typical scenario where after a merger, multiple nodes detect the merger and thus start the ring-unification algorithm. The number of nodes detecting the merger depends on the scenario; in our simulations, it was 10–15% of the total nodes. As can be seen in Figs. 2 and 3, the simple ring unification algorithm ($F = 1$) consumes minimum messages but takes maximum time when compared to different variations of

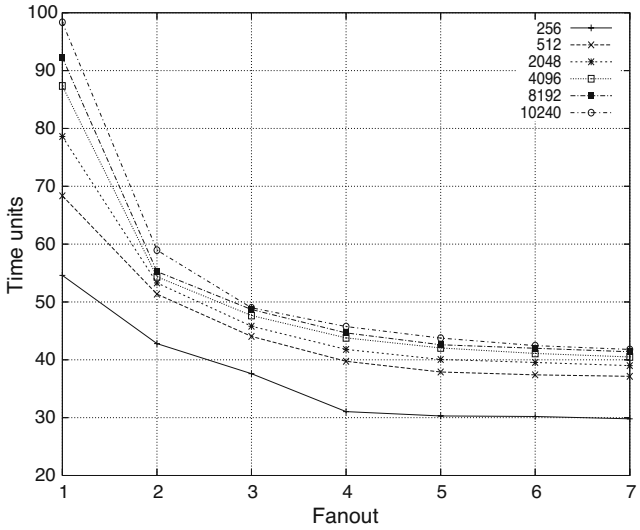

Fig. 2 Evaluation of time complexity for a typical scenario with multiple nodes detecting the merger for various network sizes and fanouts

the gossip-based ring unification algorithm. For higher values of F, the time complexity decreases while the message complexity increases. Increasing the fanout after a threshold value (around 3–4 in this case) will not considerably decrease the time complexity, but will just generate many unnecessary messages.

To proper understand the performance of the proposed algorithm, we generated scenarios where only one node would start the merger of the two rings. We randomly select, with uniform probability, the two

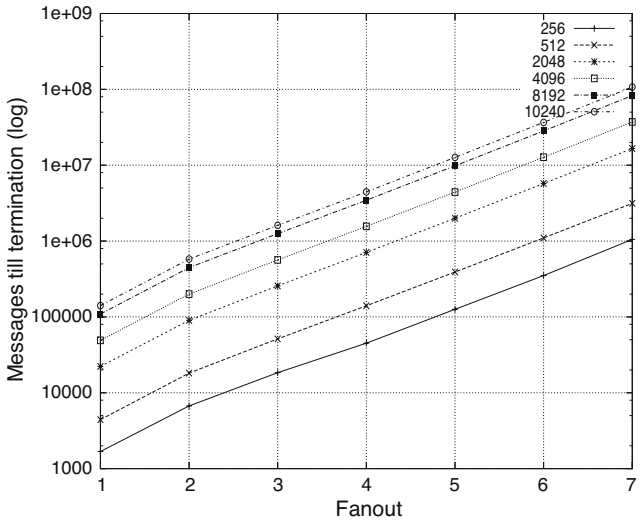

Fig. 3 Evaluation of message complexity for a typical scenario with multiple nodes detecting the merger for various network sizes and fanouts

nodes that are involved in the merger, i.e. the node p that detects the merger and the node that p detects from its passive list. Hence, the distance between them on the ring varies. For our experiments, each of the two rings had approximately half of the total number of nodes in the system before the merger. We choose the rate of checking *detqueue* to be every five time units and the rate of periodic stabilization (PS) to be every ten time units. The motivation for choosing a lower PS rate is to study the performance of the ring unification algorithm with minimum influence from PS.

We simulated ring unification for various network sizes of powers of 2 to study its scalability. Figure 4 shows the time complexity for varying network sizes. The x-axis is on a logarithmic scale, while the y-axis is linear. The graph for the gossip-based algorithms is linear, which suggests a $O(\log n)$ time complexity. In contrast, the simple ring unification graph (F = 1) is exponential, indicating that it does not scale well, i.e. $\omega(\log n)$ time complexity. In Fig. 5, we plot the number of ring unification messages sent by each node during the merger, i.e. the total number of messages induced by the algorithm until termination divided by the number of nodes. The linear graph on a log-log plot indicates a polynomial messages complexity. As expected, the number of messages per node grows slower for simple ring unification compared to gossip-based ring unification.

Figure 6 illustrates the tradeoff between time and message complexity. It shows that the goals of decreasing time and message complexity are conflicting. Thus, to decrease the number of messages, the time for completion will increase. Similarly, opting for convergence in lesser time will generate more messages. A suitable fanout value can be used to adapt the ring unification algorithm according to the requirements and network infrastructure available.

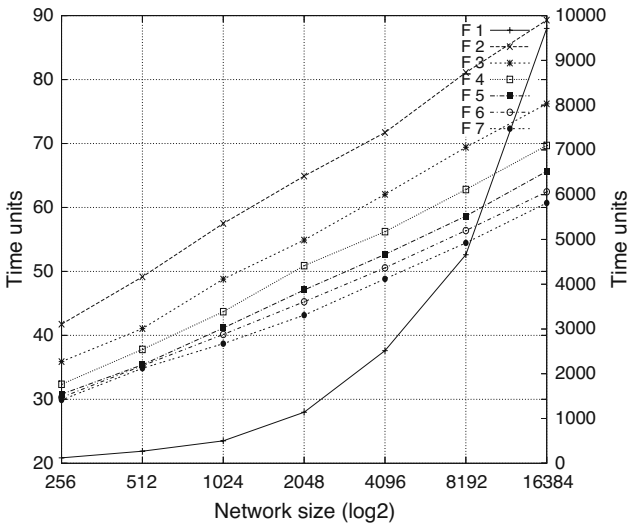

Fig. 4 Evaluation of time complexity when only one node starts the merger. Only F 1 is plotted against the right y-axis

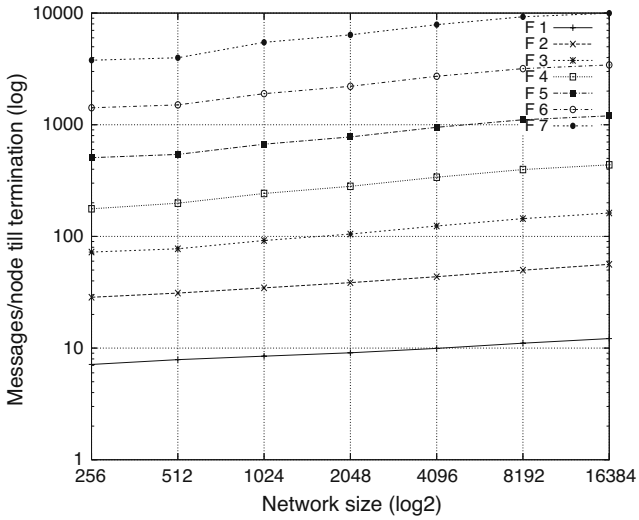

Fig. 5 Evaluation of message complexity per node when only one node starts the merger

For the rest of the evaluations, we use a worst case scenario where only a single node detects the merger.

Next, we evaluated the time and message complexity for a network to converge to a *strongly stable* ring from a *loopy* state of two cycles. As defined by Liben-Nowell et al. [25], a Chord network is *weakly stable* if, for all nodes u, $(u.succ).pred = u$ and *strongly stable* if, in addition, for each node u, there is no node v such that

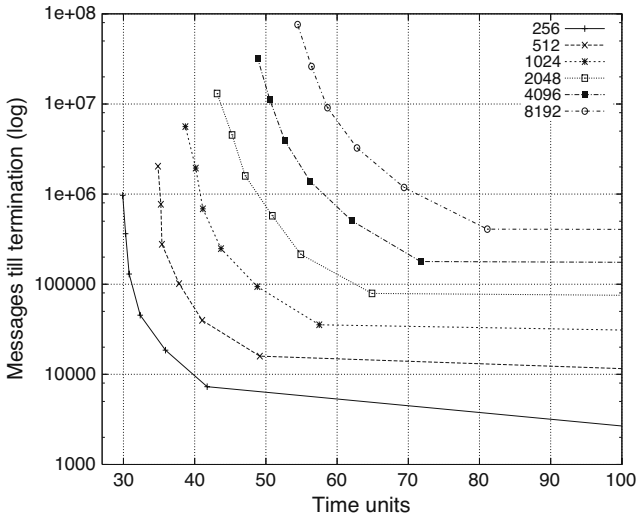

Fig. 6 Tradeoff between time and message complexity

$u < v < u.succ$. A loopy network is one which is weakly but not strongly stable. The scenario for the simulations was to create a loop of two cycles from one-fifth of the total number of nodes. Thereafter, we generated events of node joins for the remaining four-fifth nodes at an exponentially distributed inter-arrival rate. As in all experiments, the identifiers of the joining nodes were generated randomly with uniform probability. Thus, the nodes joined at different points in the loop. We then made one random node detect the loop by discovering a random node from the other cycle, triggering the ring unification algorithm. Figures 7 and 8 show the time and message complexity for the loopy network to converge to a strongly stable ring. The figures depict the effect of fanout on time and message complexity.

We evaluated rings unification under churn, *i.e.* nodes join and fail during the merger. Since we are using a scenario where only one node detects the merger, with low probability, the algorithm may fail to complete and the merged overlay may not converge under churn, especially for simple ring unification. The reason being intuitive: for simple unification, the two MLOOKUPS generated by the node detecting the merger while traveling through the network may fail as the node forwarding the MLOOKUP may fail under churn. With higher values of F and in typical scenarios where multiple nodes detect the merger, the algorithm becomes more robust to churn as it creates multiple MLOOKUPS. The results presented in Figs. 9 and 10 are only when the rings successfully converge. For simulation, after a merge event, we generate events of joins and fails until the unification algorithm terminates. With high churn, we mean that the inter-arrival time between events of joins and fails is less, thus representing highly dynamic conditions. Choosing a high inter-arrival time between events will create less joins and fails and thus churn will be less. For the simulations presented here, we choose inter-arrival time between events of joins and failures to be 30 units for high churn and 45 units for low churn, and an equal probability for a event to be a join or a fail. Figures 9 and 10 show how different values of F affect the convergence of the rings under different levels

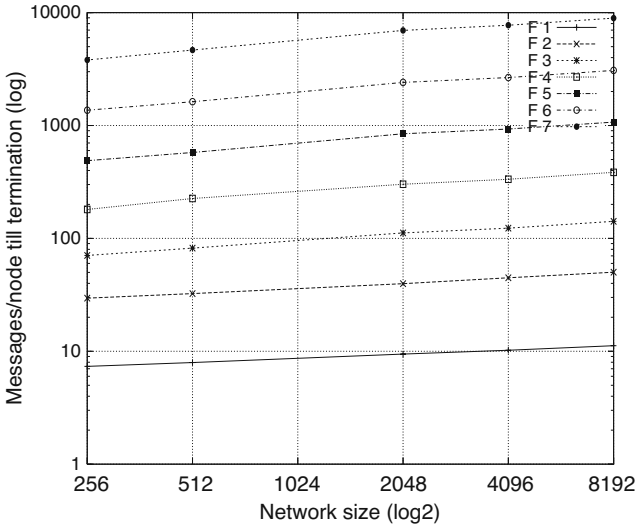

Fig. 8 Evaluation of message complexity for a loopy network

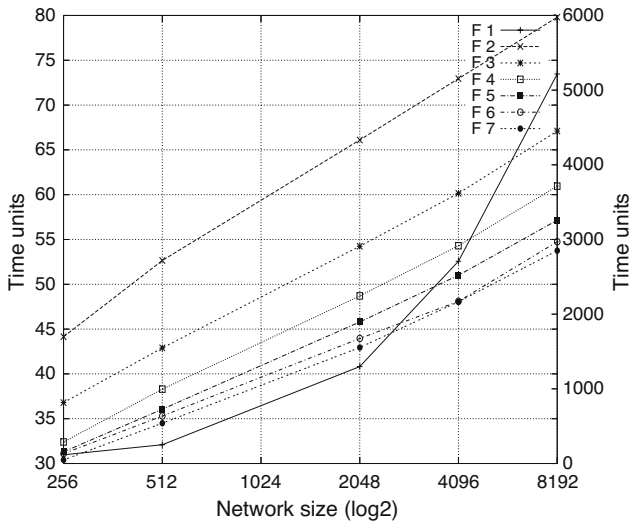

Fig. 7 Evaluation of time complexity for a loopy network. Only F 1 is plotted against the right y-axis

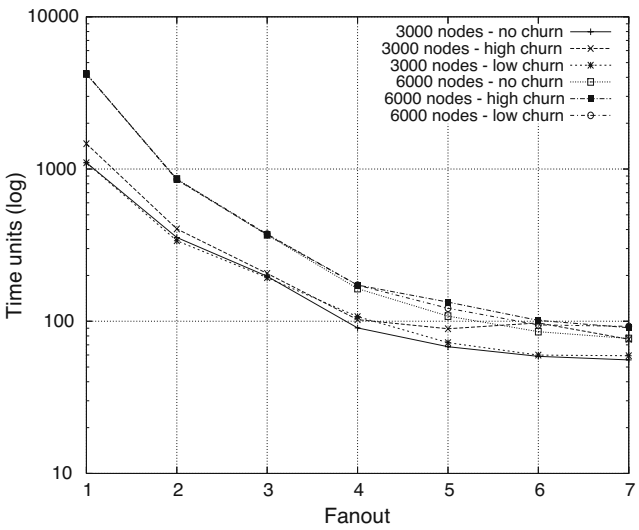

Fig. 9 Evaluation of time complexity under churn

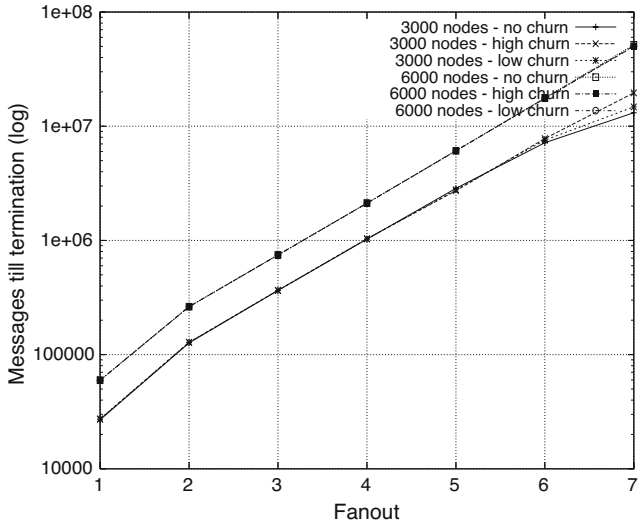

Fig. 10 Evaluation of message complexity under churn

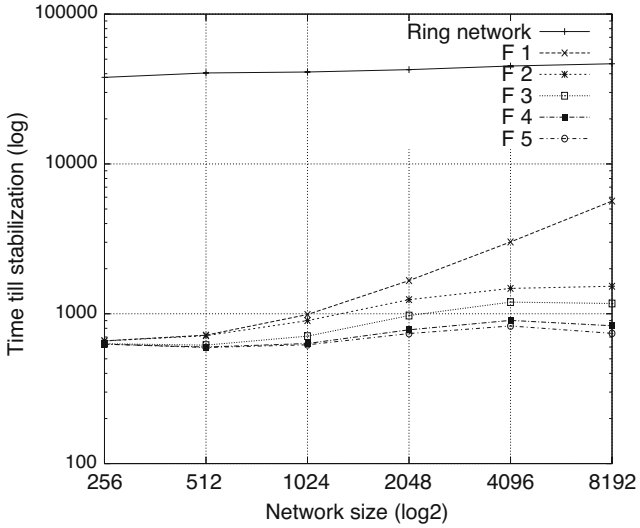

Fig. 11 Comparison of time complexity of ring unification and SSRN

of churn, mainly showing the algorithm works under churn without affecting message and time complexity much.

Further, we simulated the algorithms under churn to see how often they do not converge to a ring. We ran experiments with 200 different seeds for sizes ranging from 256 to 2048 nodes. We considered an execution successful if 95% of the nodes had correct successor pointers, as all successor pointers can not be correct while nodes are joining and failing. Thereafter, the remaining pointers are updated by Chord's periodic stabilization. For the 200 executions, we observed only 1 unsuccessful execution for network size 1024 and 2 unsuccessful executions for network size 2048. The unsuccessful executions happened only for simple ring unification, while executions with gossip based ring unification were always successful. Even for the unsuccessful executions, given enough time, PS updates the successor pointers to correct values.

We compared our algorithm with a Self-Stabilizing Ring Network (SSRN) [36] protocol. The results of our simulations comparing time and message complexity for various network sizes for the two algorithms have been presented in Figs. 11 and 12, depicting that ring unification consumes lesser time and messages compared to SSRN. The main reason for the better performance of our algorithm is that it has been designed specifically for merging rings. On the other hand, SSRN is a non-terminating algorithm that runs in the background like PS to find closer nodes. As evaluated previously, simple ring unification (fanout = 1) does not scale well for time complexity, which can be seen in Fig. 11.

Finally, we evaluate the scenario where a node may falsely detects a merger. Figure 13 shows the message complexity of the algorithm in case of a false detection. As can be seen, for lower fanout values, the message complexity is less. Even for higher fanouts, the number of messages generated are acceptable, thus showing that the algorithm is lean. We believe this to be important as most SONs do not have perfect failure detectors, and hence can give rise to inaccurate suspicions.

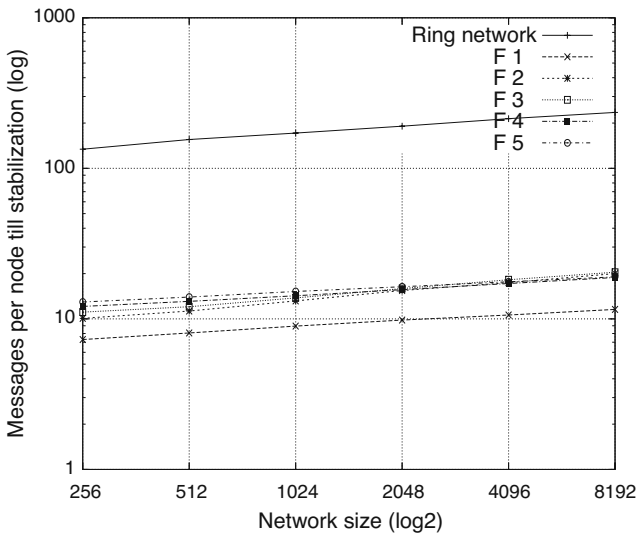

Fig. 12 Comparison of message complexity of ring unification and SSRN

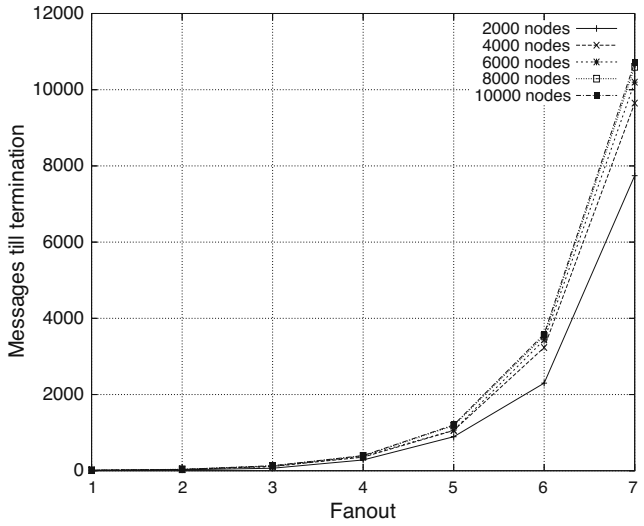

Fig. 13 Evaluation of message complexity in case a node falsely detects a merger for various network sizes and fanouts

Our simulations show that a fanout value of 3–4 is good for a system with several thousand nodes, even with respect to churn and false-positives.

5 Related work

Much work has been done to study the effects of churn on a structured overlay network [27], showing how overlays can cope with massive node joins and failures, thus showing how overlays are resilient to partitions. Datta et al. [8] have presented the challenges of merging two overlays, claiming that ring-based networks cannot operate until the merger operation completes. In contrast, we show how unification can work under churn while the merger operation is not complete. In a followup work, Datta et al. [9] show how to merge two P-Grid [2] SONs. There work differs from ours as P-Grid is a tree-based SON, while we focus on ring-based SONs.

The problem of constructing a SON from a random graph is, in some respects, similar to merging multiple SONs after a network merger, as the nodes may get randomly connected after a partition heals. Shaker et al. [36] have presented a ring-based algorithm for nodes in arbitrary state to converge into a directed ring topology. Their approach is different from ours, in that they provide a non-terminating algorithm which should be used to replace all join, leave, and failure handling of an existing SON. Replacing the topology maintenance algorithms of a SON may not always be feasible, as SONs may have intricate join and leave procedures to guarantee lookup consistency [14, 24, 26]. In contrast, our algorithm is a terminating algorithm that works as a plug-in for an already existing SON.

Kunzmann et al. [21] have proposed methods to improve the robustness of SONs. They propose to use a bootstrapping server to detect a merger by making the peer with the smallest identifier to send periodic messages to the bootstrap server. As soon as the bootstrap server receives messages from different peers, it will detect the existence of multiple rings. Thereafter, all the nodes have to be informed about the merger. While their approach has the advantage of having minimum false detections, it depends on a central bootstrap server. They lack a full algorithm and evaluation of how the merger will happen. Evaluation of the merge detection process and informing all peers about the detection is also missing.

Montresor et al. [29] show how Chord [39] can be created by a gossip-based protocol [18]. However, their algorithm depends on an underlying membership service like Cyclon [41], Scamp [13] or Newscast [19]. Thus the underlying membership service has to first cope with network mergers (a problem worth studying in its own right), where after T-Chord can form a Chord network. We believe one needs to investigate further how these protocols can be combined, and their epochs be synchronized, such that the topology provided by T-Chord is fed back to the SON when it has converged. Though the general performance of T-Chord has been evaluated, it is not known how it performs in the presence of network mergers when combined with various underlying membership services.

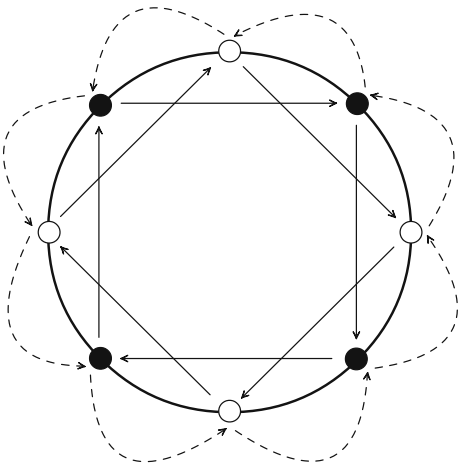

Fig. 14 A case where chord and the ring network protocol would break a connected graph into two components. *Lines* represent successor pointers while *dashed lines* represent a finger

As we show below, it might happen that an initially connected graph can be split into two separate components by the Chord [39] and SSRN [36] protocols. This scenario is a counter-proof of the claim that SSRN is self-stabilizing. Consider a network which consists of two perfect rings, yet the nodes have fingers pointing to nodes in the other ring. This can easily happen in case of unreliable failure detectors [7] or networks partitions. Normally, the PS rate is higher than fixing fingers, thus due to a temporary partition, it might happen that nodes update their successor pointers, yet before they fix their fingers, the partition heals. In such a scenario, SSRN splits the connected graph into two separate partitions, thus creating a partition of the overlay, while the underlay remains connected. An example of such a scenario is shown in Fig. 14, where the filled circles are nodes that are part of one ring and the empty circles are nodes that are part of the other ring. Each node has one finger pointing to a node in the other ring. The fix-finger algorithm in Chord updates the fingers by making lookups. In this case, a lookup will always return a node in the same ring as the one making the lookup. Consequently, the finger pointing to the other ring will be lost. Similarly, the pointer jumping algorithm used by SSRN to update its fingers will also drop the finger pointing to a node in the other ring. On the contrary, the ring-unification algorithm proposed in this paper will fix such a graph and converge it to a single ring.

Some SONs employ the ring based identifier space, which they mix with a prefix-based tree [33]. For example in Pastry [34], a responsible node for an identifier is the node with numerically closest identifier and the lookups are forwarded to nodes sharing the longest prefix with the identifier being looked up. Our algorithm can be modified for use by such SONs by replacing the closestpreceedingnode-procedure with the equivalent for the employed SON. The trymerge-procedure does not have to be changed since updating the predecessor and successor is similar to recording nodes with identifiers closest to a node.

The problem of network partitions and mergers has been studied in other distributed systems, such as in distributed databases [10] and distributed file systems [40]. These studies focus on problems created by the partition and merger on the data level, while our focus is on the routing level. We believe that such ideas, if combined with algorithms such as those we propose, can be used for handling data updates on SONs. That is, nevertheless, outside the scope of this paper.

The results of this paper extend on our previous work [35] by also considering loopy networks, the SSRN protocol [36], and including additional experimental results.

6 Conclusion

We have argued that the problem of partitions and mergers in structured peer-to-peer systems, when the underlying network partitions and recovers, is of crucial importance. We have presented a simple and a gossip-based algorithm for merging similar ring-based structured overlay networks after the underlying network merges.

Though we believe that the problem of dealing with network mergers is crucial, we think that such events happen more rarely. Hence, it might be justifiable in certain application scenarios that a slow paced algorithm runs in the background, consuming little resources, while ensuring that any potential problems with partitions will eventually be rectified. In such scenarios, our simple ring unification is more suitable. If on the other hand, one would prefer to speed up the unification process by consuming more messages, our gossip-based ring unification is more suitable. We have shown how the algorithm can be tuned to achieve a tradeoff between the number of messages consumed and the time before the overlay converges. We have evaluated our solution in realistic dynamic conditions, and showed that with high fanout values, the algorithm can converge quickly under churn. We have also shown that our solution generates few messages even if a node falsely starts the algorithm in an already converged SON. Finally, we have shown that our algorithm recovers from pathological scenarios, such as loopy rings, which might result from network partitions.

We tried many variations of the algorithms before reaching those that are reported in this paper. Initially, we had an algorithm that was not gossip-based, i.e. was not periodic and did not have any randomization. Albeit the algorithm was quite fast, it heavily overconsumed messages, making it infeasible for a large scale network. For that reason, we added the fanout parameter, and made it run periodically. Without randomization, we could construct pathological scenarios, in which that algorithm would not be able to merge the rings.

Future work We believe that dealing with partitions and mergers is a small part of a bigger, and more important, goal: making SONs that can recover from any configuration. We believe that it is desirable to make a self-stabilizing ring algorithm, which can be proved

to recover from all possible states, including loopy and partitioned while consuming minimum time and messages.

We believe that it is interesting to investigate whether gossip-based topology generators, such as T-man [18] and T-chord [29], can be used to handle network mergers. These services, however, make use of an underlying membership service, such as Cyclon [41], Scamp [13], or Newscast [19]. Hence, one has to first investigate how well such membership services recover from network partitions (we believe this to be interesting in itself). Thereafter, one can explore how such topology generators can be incorporated into a SON.

Mathematical analysis of gossip-protocols is often done through simple recurrence relations or by using Markov chains, where the state of the chain can be the number of infected nodes [12]. The algorithms we have proposed mix deterministic DHT algorithms with that of gossip protocols. Consequently, we believe that an analysis of our algorithms will require modelling the routing pointers of every node as part of the chain state. We solicit such an analysis and believe it is an interesting future direction for this research.

Acknowledgements This research has been funded by the European Projects SELFMAN and EVERGROW, VINNOVA 2005-02512 TRUST-DIS, and SICS Center for Networked Systems (CNS).

References

1. Aberer K, Alima LO, Ghodsi A, Girdzijauskas S, Haridi S, Hauswirth M (2005) The essence of P2P: a reference architecture for overlay networks. In: Proceedings of the 5th international conference on peer-to-peer computing (P2P'05). IEEE Computer Society, Los Alamitos, pp 11–20, August
2. Aberer K, Cudré-Mauroux P, Datta A, Despotovic Z, Hauswirth M, Punceva M, Schmidt R (2003) P-grid: a self-organizing structured P2P system. SIGMOD Rec 32(3): 29–33
3. Alima LO, Ghodsi A, Haridi S (2004) A framework for structured peer-to-peer overlay networks. In: Post-proceedings of global computing. Lecture notes in computer science (LNCS), vol 3267. Springer, Berlin Heidelberg New York, pp 223–250
4. Bharambe AR, Agrawal M, Seshan S (2004) Mercury: supporting scalable multi-attribute range queries. In: Proceedings of the ACM SIGCOMM 2004 symposium on communication, architecture, and protocols. ACM, Portland, pp 353–366, March
5. Brewer E (2000) Towards robust distributed systems. Invited talk at the 19th annual ACM symposium on principles of distributed computing (PODC'00)
6. Jahanian F, Labovitz C, Ahuja A (1998) Experimental study of internet stability and wide-area backbone failures. Technical report CSE-TR-382-98, University of Michigan, November
7. Chandra TD, Toueg S (1996) Unreliable failure detectors for reliable distributed systems. J ACM 43(2):225–267
8. Datta A, Aberer K (2006) The challenges of merging two similar structured overlays: a tale of two networks. In: Proceedings of the first international workshop on self-organizing systems (IWSOS'06). Lecture notes in computer science (LNCS), vol 4124. Springer, Berlin Heidelberg New York, pp 7–22
9. Datta A (2007) Merging intra-planetary index structures: decentralized bootstrapping of overlays. In: Proceedings of the first international conference on self-adaptive and self-organizing systems (SASO 2007). IEEE Computer Society, Boston, pp 109–118, July
10. Davidson SB, Garcia-Molina H, Skeen D (1985) Consistency in a partitioned network: a survey. ACM Comput Surv 17(3):341–370
11. Demers A, Greene D, Hauser C, Irish W, Larson J, Shenker S, Sturgis H, Swinehart D, Terry D (1987) Epidemic algorithms for replicated database maintenance. In: Proceedings of the 7th annual ACM symposium on principles of distributed computing (PODC'87). ACM, New York, pp 1–12
12. Eugster P Th, Guerraoui R, Handurukande SB, Kouznetsov P, Kermarrec A-M (2003) Lightweight probabilistic broadcast. ACM Trans Comput Syst 21(4):341–374
13. Ganesh AJ, Kermarrec A-M, Massoulié L (2001) SCAMP: peer-to-peer lightweight membership service for large-scale group communication. In: Proceedings of the 3rd international workshop on networked group communication (NGC'01). Lecture notes in computer science (LNCS), vol 2233. Springer, London, pp 44–55
14. Ghodsi A (2006) Distributed *k*-ary system: algorithms for distributed hash tables. PhD dissertation, KTH—Royal Institute of Technology, Stockholm, December
15. Gilbert S, Lynch NA (2002) Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM Spec Interest Group Algorithms Comput Theory News 33(2):51–59
16. Gummadi K, Gummadi R, Gribble S, Ratnasamy S, Shenker S, Stoica I (2003) The impact of DHT routing geometry on resilience and proximity. In: Proceedings of the ACM SIGCOMM 2003 symposium on communication, architecture, and protocol. ACM, New York, pp 381–394
17. Harvey N, Jones MB, Saroiu S, Theimer M, Wolman A (2003) Skipnet: a scalable overlay network with practical locality properties. In: Proceedings of the 4th USENIX symposium on internet technologies and systems (USITS'03). USENIX, Seattle, March
18. Jelasity M, Babaoglu Ö (2005) T-man: gossip-based overlay topology management. In: Proceedings of 3rd workshop on engineering self-organising systems (EOSA'05). Lecture notes in computer science (LNCS), vol 3910. Springer, Berlin Heidelberg New York, pp 1–15
19. Jelasity M, Kowalczyk W, van Steen M (2003) Newscast computing. Technical report IR–CS–006, Vrije Universiteit, November
20. Kaashoek MF, Karger, DR (2003) Koorde: a simple degree-optimal distributed hash table. In: Proceedings of the 2nd interational workshop on peer-to-peer systems (IPTPS'03). Lecture notes in computer science (LNCS), vol 2735. Springer, Berkeley, pp 98–107

21. Kunzmann G, Binzenhöfer A (2006) Autonomically improving the security and robustness of structured P2P overlays. In: Proceedings of the international conference on systems and networks communications (ICSNC 2006). IEEE Computer Society, Tahiti, October–November
22. Leong B, Liskov B, Demaine E (2004) EpiChord: parallelizing the chord lookup algorithm with reactive routing state management. In: 12th international conference on networks (ICON'04). IEEE Computer Society, Singapore, November
23. Li J, Stribling J, Morris R, Kaashoek MF (2005) Bandwidth-efficient management of DHT routing tables. In: Proceedings of the 2nd USENIX symposium on networked systems design and implementation (NSDI'05). USENIX, Boston, May
24. Li X, Misra J, Plaxton, CG (2004) Brief announcement: concurrent maintenance of rings. In: Proceedings of the 23rd annual ACM symposium on principles of distributed computing (PODC'04). ACM, New York, p 376
25. Liben-Nowell D, Balakrishnan H, Karger DR (2002) Observations on the dynamic evolution of peer-to-peer networks. In: Proceedings of the first international workshop on peer-to-peer systems (IPTPS'02). Lecture notes in computer science (LNCS), vol 2429. Springer, Berlin Heidelberg New York
26. Lynch NA, Malkhi D, Ratajczak, D (2002) Atomic data access in distributed hash tables. In: Proceedings of the first interational workshop on peer-to-peer systems (IPTPS'02). Lecture notes in computer science (LNCS). Springer, London, pp 295–305
27. Mahajan R, Castro M, Rowstron A (2003) Controlling the cost of reliability in peer-to-peer overlays. In: Proceedings of the 2nd international workshop on peer-to-peer systems (IPTPS'03). Lecture notes in computer science (LNCS), vol 2735. Springer, Berkeley, pp 21–32
28. Manku GS, Bawa M, Raghavan P (2003) Symphony: distributed hashing in a small world. In: Proceedings of the 4th USENIX symposium on internet technologies and systems (USITS'03). USENIX, Seattle, March
29. Montresor A, Jelasity M, Babaoglu Ö (2005) Chord on demand. In: Proceedings of the 5th international conference on peer-to-peer computing (P2P'05). IEEE Computer Society, Los Alamitos, August
30. PINR (2008) Taiwan earthquake on December 2006. http://www.pinr.com/report.php?ac=view_report&report_id=602. Accessd January 2008
31. Oppenheimer D, Ganapathi A, Patterson DA (2003) Why do internet services fail, and what can be done about it? In: USITS'03: proceedings of the 4th conference on USENIX symposium on internet technologies and systems. USENIX Association, Berkeley, pp 1–1
32. Paxson V (1997) End-to-end routing behavior in the internet. IEEE/ACM Trans Netw (TON) 5(5):601–615
33. Plaxton CG, Rajaraman R, Richa, AW (1997) Accessing nearby copies of replicated objects in a distributed environment. In: Proceedings of the 9th annual ACM symposium on parallelism in algorithms and architectures (SPAA'97). ACM, New York, pp 311–320
34. Rowstron A, Druschel P (2001) Pastry: scalable, distributed object location and routing for large-scale peer-to-peer systems. In: Proceedings of the 2nd ACM/IFIP international conference on middleware (MIDDLEWARE'01). Lecture notes in computer science (LNCS), vol 2218. Springer, Heidelberg, pp 329–350, November
35. Shafaat TM, Ghodsi A, Haridi S (2007) Handling network partitions and mergers in structured overlay networks. In: Proceedings of the 7th international conference on peer-to-peer computing (P2P'07). IEEE Computer Society, Los Alamitos, pp 132–139, September
36. Shaker A, Reeves DS (2005) Self-stabilizing structured ring topology P2P systems. In: Proceedings of the 5th international conference on peer-to-peer computing (P2P'05). IEEE Computer Society, Los Alamitos, pp 39–46, August
37. SicsSim (2008) http://dks.sics.se/p2p07partition/. Accessed January 2008
38. Stoica I, Morris R, Liben-Nowell D, Karger DR, Kaashoek MF, Dabek F, Balakrishnan H (2002) Chord: a scalable peer-to-peer lookup service for internet applications. Technical report TR-819, Massachusetts Institute of Technology (MIT), January
39. Stoica I, Morris R, Liben-Nowell D, Karger DR, Kaashoek MF, Dabek F, Balakrishnan H (2003) Chord: a scalable peer-to-peer lookup protocol for internet applications. IEEE/ACM Trans Netw (TON) 11(1):17–32
40. Terry DB, Theimer M, Petersen K, Demers AJ, Spreitzer M, Hauser C (1995) Managing update conflicts in Bayou, a weakly connected replicated storage system. In: Proceedings of the 15th ACM symposium on operating systems principles (SOSP'95). ACM, New York, pp 172–183, December
41. Voulgaris S, Gavidia D, van Steen M (2005) Cyclon: inexpensive membership management for unstructured p2p overlays. J Netw Syst Manag 13(2):197–217
42. Waldspurger CA, Weihl WE (1994) Lottery scheduling: flexible proportional-share resource management. In: Proceedings of the first symposium on operating systems design and implementation (OSDI'94). USENIX, Seattle, pp 1–11, November

Tallat M. Shafaat is a PhD candidate at KTH - Royal Institute of Technology, Sweden and a member of CSL at SICS - Swedish Institute of Computer Science. Earlier, he completed his B.Sc. at GIK Institute of Engineering Sciences and Technology, Pakistan and an M.Sc. at KTH - Royal Institute of Technology. His research interests include large-scale distributed systems, distributed algorithms and peer-to-peer systems.

Ali Ghodsi is a senior researcher at the Swedish Institute of Computer Science (SICS). He got his PhD in Computer Science from KTH–Royal Institute of Technology in 2006, and an M.B.A. and an M.Sc. from Mid-Sweden University in 2002. His research interest is in distributed computing in general, and in the theory and practice of large-scale dynamic distributed systems in particular.

Seif Haridi received the Ph.D. degree in computer systems from the KTH - Royal Institute of Technology, Sweden, in 1981. He is currently the scientific leader of the Computer Systems Laboratory and the Chief Scientist of the Swedish Institute of Computer Science (SICS) in Kista, Sweden. He is also a Professor of computer systems at the School of Information and Communication Technology, the KTH - Royal Institute of Technology, Sweden.