Handling Network Partitions and Mergers in Structured Overlay Networks *

Tallat M. Shafaat†   Ali Ghodsi‡   Seif Haridi†

†Royal Institute of Technology (KTH),   ‡Swedish Institute of Computer Science (SICS)
{tallat,haridi}(at)kth.se   ali(at)sics.se

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.

1 Introduction

Structured Overlay Networks (SONs)—such as Chord [29], Pastry [26], and SkipNet [13]—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 those of distributed databases [5] and distributed file systems [30].

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, most SONs cope with network partitions, but not with network mergers. We believe that this is because a network partition, as seen from the perspective a single node, is identical to massive node failures. Since SONs have been designed to cope with churn, they can self-manage in the presence of such partitions. However, 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.* [4] 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". Birman [2] argues that ring-based SONs are inherently ill-suited for dealing with 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 [11, 3, 9].

---

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

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 [30, 6].

In this paper, we present an algorithm for merging any number of similar structured overlays. We will limit ourselves to ring-based overlays, since they constitute the majority of the SONs. It is desirable that a solution to the problem of merging rings takes minimum amount of time to complete (time complexity). At the same time, it is desirable that the solution has a minimal bandwidth consumption (message and bit complexity). These two goals are conflicting, as shown by the following two extreme cases. On the one hand, it is possible to construct an algorithm that completes in minimal time by having all the nodes repeatedly spreading all their routing information to every other node through an overlay broadcast [7, 10, 9]. On the other hand, it is possible to construct an algorithm which tries to minimize the bandwidth consumption by passing a "merging" token around each of the rings. Hence, it is desirable to find an algorithm which strikes a balance between time, bit, and message complexity.

The contribution of this paper is a ring merging algorithm, which allows the system designer to adjust, through a *fanout* parameter, the tradeoff between message complexity and 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 widely believed to be difficult [2] or impossible [4]. 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.

Outline   Section 2 serves as a background by motivating and defining our choice of ring-based SONs. Section 3 introduces the *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.

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 Ring Geometry   The reason for confining ourselves to ring-based SONs is twofold. First, ring-based SONs constitute a majority of the SONs, including Chord [29], Pastry [26], SkipNet [13], DKS [9], Koorde [16], Viceroy [23], Mercury [1], Symphony [24], Epi-Chord [17], and Accordion [18]. Second, Gummadi *et al.* [12] diligently compared the geometries of different SONs, 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.

Our results apply to all ring-based SONs. Nevertheless, we assume a SON similar to Chord [29] to simplify the understanding of our algorithms.

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 $\log_2(\mathcal{N})$.

Ring-based SONs also maintain additional routing pointers on top of the ring to enhance routing. To keep things 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 $0 < i < \log_2(\mathcal{N})$. 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 $succ$ node 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 [20], it also recovers from network partitions, making each component of the partition eventually form its own ring. Our simulation results confirm this, though they are omitted due to space constraints. The problem that remains unsolved, which is the focus of the rest of the paper, is how several independent rings efficiently can 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, in its passive list. Every node periodically pings nodes in its passive list to detect if a failed node is again alive. 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. It might also be the case that a node with the same address and identifier as a failed node joins the ring. The ring merging algorithm should be able to cope with the first case, by trying to ensure that such false-positives will terminate the algorithm quickly. The latter case can be dealt with by associating with every node a globally unique random nonce, which is generated each time a node joins the network. Hence, a new node can always be differentiated from an old node with the same 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. 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. 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** $detqueue \neq \emptyset$ **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** $id \neq n$ **and** $id \neq succ$ **then**
8:         **if** $id \in (n, succ)$ **then**
9:             **sendto** id : TRYMERGE($n, succ$)
10:        **else if** $id \in (pred, n)$ **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** $csucc \in (n, succ)$ **then**
20:         $succ := csucc$
21:     **end if**
22:     **sendto** n : MLOOKUP($cpred$)
23:     **if** $cpred \in (pred, n)$ **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
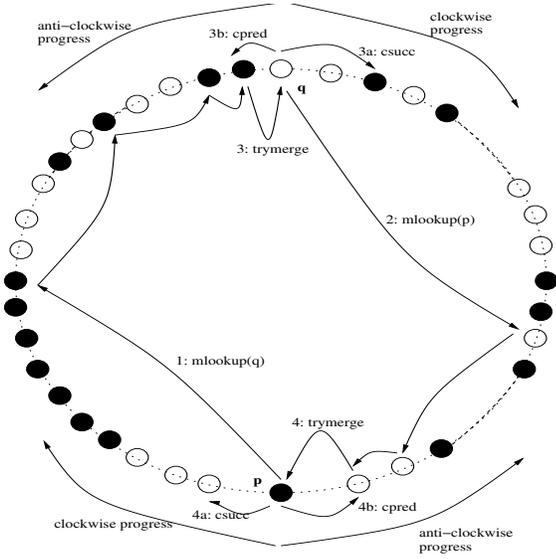
Figure 1: Filled circles belong to SON1 and empty circles belong to SON2. The algorithm starts when p detects q, p makes an MLOOKUP to q and asks q to make an MLOOKUP to p.

the event TRYMERGE($cpred, csucc$), which tries to do the actual merging by updating $succ$ and $pred$ pointers.

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 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.

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 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. Also, 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 $detqueue \neq \emptyset$ at p**
2:     $\langle q, f \rangle := detqueue.$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** $id \neq n$ **and** $id \neq succ$ **then**
8:         **if** $f > 1$ **then**
9:             $f := f - 1$
10:             $r :=$ randomnodeinRT()
11:             **at r :** $detqueue.$enqueue($\langle id, f \rangle$)
12:         **end if**
13:         **if** $id \in (n, succ)$ **then**
14:             **sendto id :** TRYMERGE($n, succ$)
15:         **else if** $id \in (pred, n)$ **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** $csucc \in (n, succ)$ **then**
25:         $succ := csucc$
26:     **end if**
27:     **sendto n :** MLOOKUP($cpred, F$)
28:     **if** $cpred \in (pred, n)$ **then**
29:         $pred := cpred$
30:     **end if**
31: **end event**

Algorithm 2 is, as its name suggests, partly gossip-based. The algorithm is essentially the same as the simple ring unification algorithm, but it starts multiple such mergers at

random places on the rings. The basic idea is to augment MLOOKUP(id), such 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 used to avoid the algorithm to consume too many messages, and therefore give rise to positive feedback cycles which 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 only is 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.

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. If not said otherwise, message complexity is until termination, while time complexity is until completion.

The evaluations are done in a stochastic discrete event simulator [28] 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 18 runs with different random seeds.

We first evaluate the message and time complexity of the algorithms in the typical scenario where after merger, many nodes simultaneously detect alive nodes in their passive lists. A worst case scenario can be when only a single node detects the existence of another ring. Thereafter, we evaluate the performance of the algorithms while node joins and failures are taking place during the ring merging process. Finally, we evaluate message complexity of the algorithms when a node falsely believes that it has detected another ring.

Each 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, on which the simulator divides the set of nodes into as many components as requested by the partition event. A partition

event is implemented using lottery scheduling [32] 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 Section 2 and 3). Furthermore, Node join and fail events are triggered in each partitioned component. Thereafter, a network merger event simply again allows messages to reach other network components, triggering the detection of alive nodes in the passive lists, and hence starting the ring unification algorithms. For the simulations, time to send a message is exponentially distributed with mean 5 for periodic stabilization, and is 1 time unit for 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 our simulation graphs, a fanout of 1 represents the simple ring unification algorithm.
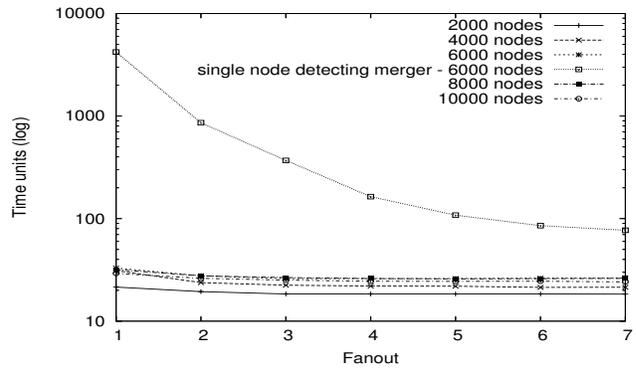

Figure 2: Evaluation of Time Complexity for a typical scenario with multiple nodes detecting the merger for various *network sizes* and *fanouts*.
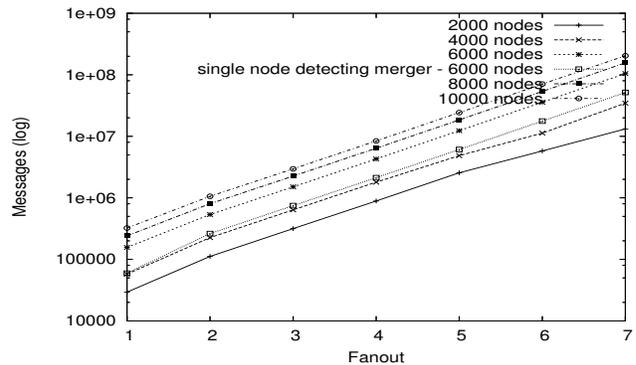

Figure 3: Evaluation of Message Complexity for a typical scenario with multiple nodes detecting the merger for various *network sizes* and *fanouts*.

Figure 2 and 3 show the time and message complexity for a typical scenario where after a merger, multiple nodes
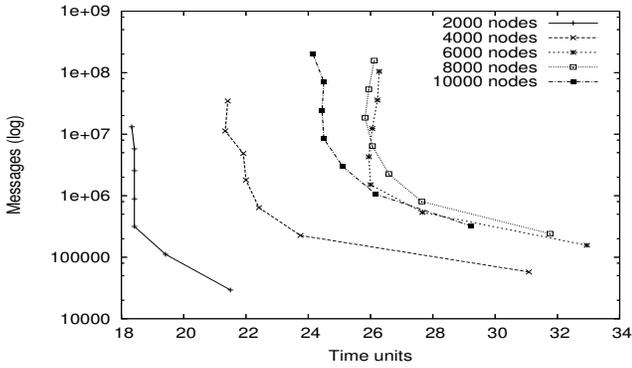
Figure 4: Comparing Time and Message complexity where multiple nodes detect the merger for various *network sizes* and *fanouts*.

detect the merger and thus start the ring-unification algorithm. The number of nodes detecting the merger depend on the scenario, in our simulations, it was 25-35% of the total nodes. The evaluation shows that while the time complexity is less for multiple nodes detecting compared to a single node detecting the merger, the message complexity is more when multiple nodes detect the merger.

As can be seen in Figures 2 and 3, the simple ring unification algorithm ($F = 1$) consumes minimum messages but takes maximum time. 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 messages. Figure 4 shows a tradeoff between time complexity and message complexity. Choosing to have less time for completion of mergers will create more messages, and vice versa. As can be seen from Figure 3, the algorithm generates a lot of messages before termination, though the completion property might have been satisfied earlier. We would like to further explore optimizations to reduce the number of messages sent.

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

Next, we evaluate 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 a very low probability, the algorithm may fail to complete and the merged overlay may not converge under churn, especially for simple ring unification and low fanouts. 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 Figure 6 and 5 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. Figure 6 and 5 show how different values of F affect the convergence of the rings under different levels of churn, mainly showing the algorithm works under churn without effecting message and time complexity much.
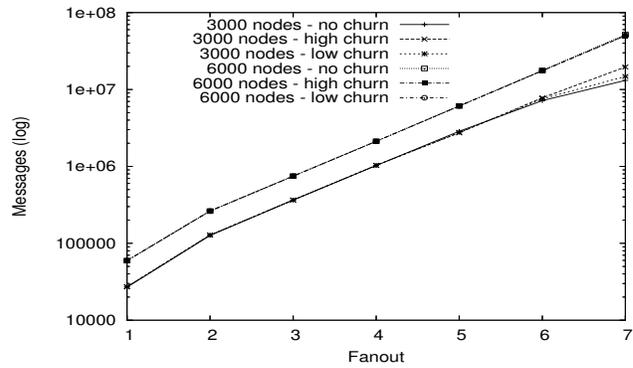

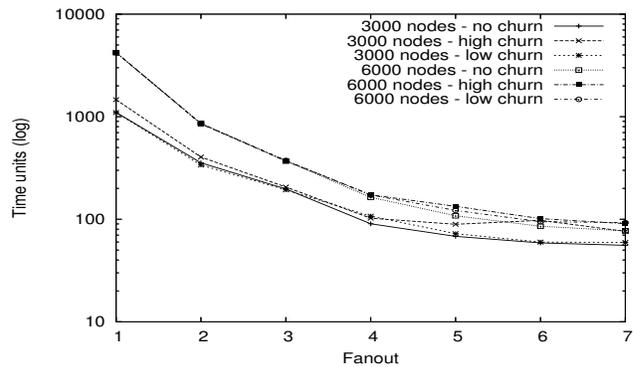**Figure 5: Evaluation of Time Complexity under churn**

Figure 6: Evaluation of Message Complexity under churn

Finally, we evaluate the scenario where a node may falsely detect a merger. Figure 7 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.
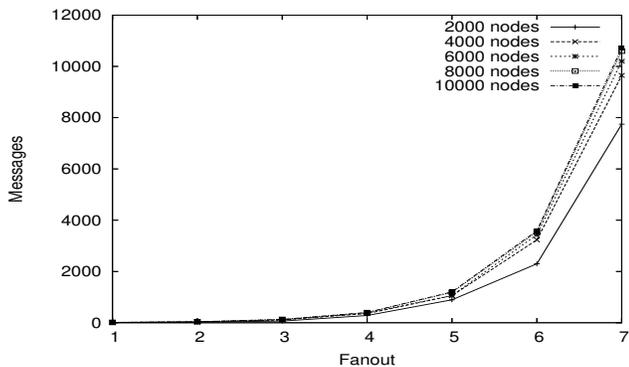
Figure 7: Evaluation of Message Complexity in case a node falsely detects a merger for various *network sizes* and *fanouts*.

Our simulations show that a fanout value of 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 [22], showing how overlays can cope with massive node joins and failures, thus showing how overlays are resilient to partitions. Datta *et al.* [4] 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. Birman [2] argued that ring-based SONs are inherently ill-suited for dealing with network partitions, while we show how ring-based SONs can be modified to deal with partitions.

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.* [27] 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 [21, 19, 9]. In contrast, our algorithm is a terminating algorithm that works as a plug-in for an already existing SON.

Montresor *et al.* [25] show how Chord [29] can be created by a gossip-based protocol [14]. However, their algorithm depends on an underlying membership service like Cyclon [31], Scamp [8] or Newscast [15]. Thus the underlying membership service has to first cope with net-

work mergers (a problem worth studying in its own right), whereafter 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.

The problem of network partitions and mergers has been studied in other distributed systems like in distributed databases [5] and distributed file systems [30]. These studies focus on problems created by the partition and merger on the data level, while we focus on the routing level.

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. Our algorithm is quite fast compared to the basic linear solution presented by Datta *et al.* [4]. 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.

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 over-consumed 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.

References

[1] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. In *Proceedings of the ACM SIGCOMM 2004 Symposium on Communication, Architecture, and Protocols*, pages 353–366, Portland, OR, USA, March 2004. ACM Press.

[2] Ken Birman. Gossip Algorithms and Emergent Shape. Invited talk at the Workshop on Gossip-based Computer Networking at the Lorentz Center, Leiden, Netherlands, December 2006.

[3] E. Brewer. Towards Robust Distributed Systems, invited talk at the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC'00), 2000.

[4] A. Datta and K. Aberer. 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)*, volume 4124 of *Lecture Notes in Computer Science (LNCS)*, pages 7–22. Springer-Verlag, 2006.

[5] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in a partitioned network: a survey. *ACM Computing Surveys*, 17(3):341–370, 1985.

[6] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing (PODC'87)*, pages 1–12, New York, NY, USA, 1987. ACM Press.

[7] S. El-Ansary, L. O. Alima, P. Brand, and S. Haridi. Efficient Broadcast in Structured P2P Netwoks. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, volume 2735 of *Lecture Notes in Computer Science (LNCS)*, pages 304–314, Berkeley, CA, USA, 2003. Springer-Verlag.

[8] A. J. Ganesh, A.-M. Kermarrec, and L Massoulié. 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)*, volume 2233 of *Lecture Notes in Computer Science (LNCS)*, pages 44–55, London, UK, 2001. Springer-Verlag.

[9] A. Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables*. PhD dissertation, KTH—Royal Institute of Technology, Stockholm, Sweden, December 2006.

[10] A. Ghodsi, L. O. Alima, S. El-Ansary, P. Brand, and S. Haridi. Self-Correcting Broadcast in Distributed Hash Tables. In *Proceedings of the 15th International Conference, Parallel and Distributed Computing and Systems*, Marina del Rey, CA, USA, November 2003.

[11] S. Gilbert and N. A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM Special Interest Group on Algorithms and Computation Theory News*, 33(2):51–59, 2002.

[12] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proceedings of the ACM SIGCOMM 2003 Symposium on Communication, Architecture, and Protocols*, pages 381–394, New York, NY, USA, 2003. ACM Press.

[13] N. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS'03)*, Seattle, WA, USA, March 2003. USENIX.

[14] M. Jelasity and Ö. Babaoglu. T-man: Gossip-based overlay topology management. In *Proceedings of 3rd Workshop on Engineering Self-Organising Systems (EOSA'05)*, volume 3910 of *Lecture Notes in Computer Science (LNCS)*, pages 1–15. Springer-Verlag, 2005.

[15] M. Jelasity, W. Kowalczyk, and M. van Steen. Newscast Computing. Technical Report IR–CS–006, Vrije Universiteit, November 2003.

[16] M. F. Kaashoek and D. R. Karger. Koorde: A Simple Degree-optimal Distributed Hash Table. In *Proceedings of the 2nd Interational Workshop on Peer-to-Peer Systems (IPTPS'03)*, volume 2735 of *Lecture Notes in Computer Science (LNCS)*, pages 98–107, Berkeley, CA, USA, 2003. Springer-Verlag.

[17] B. Leong, B. Liskov, and E. Demaine. EpiChord: Parallelizing the Chord Lookup Algorithm with Reactive Routing State Management. In *12th International Conference on Networks (ICON'04)*, Singapore, November 2004. IEEE Computer Society.

[18] J. Li, J. Stribling, R. Morris, and M. F. Kaashoek. Bandwidth-efficient management of DHT routing tables. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'05)*, Boston, MA, USA, May 2005. USENIX.

[19] X. Li, J. Misra, and C. G. Plaxton. Brief Announcement: Concurrent Maintenance of Rings. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC'04)*, page 376, New York, NY, USA, 2004. ACM Press.

[20] D. Liben-Nowell, H. Balakrishnan, and D. R. Karger. Observations on the Dynamic Evolution of Peer-to-Peer Networks. In *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS'02)*, volume 2429 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 2002.

[21] N. A. Lynch, D. Malkhi, and D. Ratajczak. 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), pages 295–305, London, UK, 2002. Springer-Verlag.

[22] R. Mahajan, M. Castro, and A. Rowstron. Controlling the Cost of Reliability in Peer-to-Peer Overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, volume 2735 of *Lecture Notes in Computer Science (LNCS)*, pages 21–32, Berkeley, CA, USA, 2003. Springer-Verlag.

[23] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC'02)*, New York, NY, USA, 2002. ACM Press.

[24] 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 (USITS'03)*, Seattle, WA, USA, March 2003. USENIX.

[25] A. Montresor, M. Jelasity, and Ö. Babaoglu. Chord on Demand. In *Proceedings of the 5th International Conference on Peer-To-Peer Computing (P2P'05)*. IEEE Computer Society, August 2005.

[26] A. Rowstron and P. Druschel. 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)*, volume 2218 of *Lecture Notes in Computer Science (LNCS)*, pages 329–350, Heidelberg, Germany, November 2001. Springer-Verlag.

[27] A. Shaker and D. S. Reeves. Self-Stabilizing Structured Ring Topology P2P Systems. In *Proceedings of the 5th International Conference on Peer-To-Peer Computing (P2P'05)*, pages 39–46. IEEE Computer Society, August 2005.

[28] SicsSim, 2007. http://dks.sics.se/p2p07partition/.

[29] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking (TON)*, 11(1):17–32, 2003.

[30] D. B. Terry, M. Theimer, K. Petersen, A. J. Demers, M. Spreitzer, and C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 172–183. ACM Press, December 1995.

[31] S. Voulgaris, D. Gavidia, and M. van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2), 2005.

[32] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI'94)*, pages 1–11. USENIX, November 1994.