2012 IEEE Sixth International Conference on Self-Adaptive and Self-Organizing Systems

Dealing with Bootstrapping, Maintenance, and Network Partitions and Mergers in Structured Overlay Networks

Tallat M. Shafaat
KTH - Royal Institute of Technology
tallat@kth.se

Ali Ghodsi
University of California, Berkeley
alig@cs.berkeley.edu

Seif Haridi
Swedish Institute of Computer Science
seif@sics.se

***Abstract*—In the last decade, numerous structured overlay networks were proposed as a scalable infrastructure to build large-scale distributed systems under dynamic environments. These overlays were touted to be fault-tolerant and self-managing; yet, as we show in this paper, they fall short of handling some extreme scenarios they envision. These scenarios include bootstrapping, and underlying network partitions and mergers. We argue that handling such extreme scenarios is fundamental to providing a fault-tolerant and self-managing system, and thus, structured overlay networks should intrinsically be able to handle them.**

In this paper, we present *ReCircle*, an overlay algorithm that apart from performing periodic maintenance to handle churn like any other overlay, can merge multiple structured overlay networks. We show how such an algorithm can be used for decentralized bootstrapping. ReCircle does not have any extra cost during normal maintenance compared to an isolated overlay maintenance algorithm. Furthermore, the algorithm is tunable to tradeoff between bandwidth consumption and time to convergence during extreme events like bootstrapping and handling network partitions and mergers. We evaluate the algorithm extensively under various scenarios through simulation and experimentation on PlanetLab.

I. Introduction

The past decade marked the rise of peer-to-peer systems that could utilize resources available at the edge of the network. This has lead researchers to develop numerous protocols, called structured overlay networks, that provide an abstraction of a lookup service over a large number of distributed nodes (e.g. [26], [10], [15]). Such overlays were touted for being fault-tolerant and self-managing, and their ability to handle decentralized and dynamic environments, e.g. peer-to-peer systems on the Internet. As it turns out, structured overlay networks fall short of handling some extreme conditions they envisioned. These extreme conditions include network partitions and mergers, bootstrapping, and flash crowds.

Network partitions are a fact of life. Hence, any long-lived Internet-scale system is bound to come across network partitions. A variety of reasons can lead to such partitions. A WAN link failure, router failure, router misconfiguration, overloaded routers, congestion due to denial of service attacks, buggy software updates, and physical damage to network equipment can all result in network partitions [21], [6], [22], [3]. Apart from software and hardware failures, political and service provider policies can also result in network partitions. For instance, the disputation between two ISPs, *Cogent* and *Telia*, lead to network breakage for a large number of customers across the Atlantic for two weeks [2]. Similarly, due to government policies, the Internet was cut-off in Egypt for more than 24 hours resulting in the network of the whole country being partitioned away [1]. Since the vision of structured overlay networks is to provide fault-tolerance and self-management at large-scale, we believe that structured overlay networks should intrinsically be able to deal with network partitions and mergers. With the exception of SkipNet [10], overlays have generally ignored the problem of network partitions and mergers; and the approach taken by SkipNet to merge multiple overlays is not applicable to other overlays (see Section V). On the same lines, while deploying an application built on top of a structured overlay, the first major problem reported and strongly suggested to be solved by Mislov *et al.* [19] was that "a reliable decentralized system must tolerate network partitions."

Efficient bootstrapping - creating a populated structured overlay network from scratch - is yet another challenge that overlays failed to address. Overlays are limited in the rate at which new nodes can join the overlay [16], hence the time duration needed to create an overlay of a large size may be long. This approach has two drawbacks. First, the system users may have to wait for a long duration, depending on the size of the overlay, before they can use the overlay. This is undesirable for example when resources are allocated for limited duration of time, or when an overlay has to be created in an ad hoc or temporary setting. Second, limiting the rate of joins may be complicated or require central coordination, which defies the ideology of structured overlay networks as they are decentralized peer-to-peer systems. Hence, given the goal of structured overlays to be self-managing and self-organizing, we believe that they should be able to bootstrap efficiently without constraints on the size of the overlay or the rate of joins.

We strongly believe that if structured overlay networks are to realize their goal of being scalable, fault-tolerant, self-managing and self-organizing, they should inherently be able

978-0-7695-4851-7/12 $26.00 © 2012 IEEE
DOI 10.1109/SASO.2012.36

to bootstrap efficiently, and handle network partitions and mergers other than only being able to deal with moderate rates of churn.

In this paper, we present an overlay algorithm, called *ReCircle*, that is capable of *(i)* bootstrapping an overlay, *(ii)* maintaining the overlay under churn, and *(iii)* handling underlying network partitions and mergers. The algorithm builds and maintains a structured overlay with a uni-directional ring geometry, yet the underlying concepts can be extended to other geometries as well. The algorithm allows a system designer to tradeoff between bandwidth consumption and time taken for bootstrapping and merging overlays. We show this tradeoff and examine the solution through simulation and experimental evaluation on PlanetLab for various scenarios and parameter values. Furthermore, during normal operation, i.e. after bootstrapping and without underlying network partitions and mergers, the algorithm behaves similar to a general overlay maintenance algorithm without any overhead.

Outline: We begin by presenting a background in Section II. We present our solution in Section III. Next, Section IV presents a detailed evaluation of the algorithm with simulations and experiments on PlanetLab. Finally, we discuss related work in Section V and conclude in Section VI.

II. Background

In this paper, we confine ourselves to ring-based structured overlay networks, as they constitute the majority of structured overlays, e.g. Chord [26], SkipNet [10], and Accordion [15]. For simplicity, we use notations from Chord, though the ideas presented in the paper are applicable to other ring-based overlays as well.

Model of a ring overlay: An overlay 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. 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, which is the first node met going anti-clockwise direction. 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 overlays also maintain additional routing pointers on top of the ring to enhance routing. Our results do not depend on how these additional pointers are placed.

Periodic maintenance: Chord handles joins and failures using a protocol called *periodic stabilization* (PS). Failures 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 $pred$, or if the successor's $pred$ is nil. Hence, any joining node is eventually properly incorporated into the ring.

III. Solution

Bootstrapping, network partitions and mergers, and flash crowds represent extreme rates of churn. Bootstrapping and overlay mergers are similar to a large number of nodes joining the overlay simultaneously, where as network partitions are akin of massive failures. Flash crowds can be either huge number of nodes joining or leaving the overlay. Periodic stabilization (PS) can handle massive failures [17] as long as no node looses all its successor-list. Hence, if no node has its successor-list partitioned away, PS can handle network partitions, making each component of the partition eventually form its own ring. Furthermore, overlays cannot intrinsically bootstrap efficiently, handle flash crowds of joins [17], or deal with overlay mergers. To handle all these cases, we propose ReCircle, an overlay maintenance algorithm that runs periodically for normal overlay maintenance, and reacts to extreme events and starts sending messages other than the periodic messages. Periodic messages are exchanged between a node, its successor and predecessor to maintain the geometry in the node's immediate vicinity only, while the reactive messages can navigate further in the identifier space. These messages remedy the anomalies in the geometry and the overlay converges to a ring. Once the overlay converges, the reactive messages die out and the algorithm returns to act as a normal periodic maintenance algorithm.

Our methodology is different from overlay maintenance algorithms such as Chord's periodic stabilization in two aspects. First, ReCircle is reactive to extreme events, while Chord is always periodic. Being reactive is desirable for extreme events since such events invalidate several pointers simultaneously. Second, in Chord, a node periodically attempts to fix any possible anomalies in the geometry only with its immediate successor. On the other hand, as extreme events may quickly make the immediate neighbourhood of a node on the ring outdated, ReCircle is able to traverse farther away, using an operation similar to a lookup.

The solution is given as Algorithm 1. Periodically, every δ time units (line 1), each node n attempts to set its *succ* to a node clockwise closer to n than n's current successor. n

accomplishes this by retrieving its successor's $pred$ pointer, and updates $succ$ if it finds a closer successor.

Each node maintains a *queue*, which contains a list of node identifiers that represent possible (problematic) areas on the identifier space that violate the geometry of the overlay and can be fixed. These areas can arise, for example, due to churn, bootstrapping, and flash crowds. If the queue is nonempty at any node, it implies that the overlay may not be in a converged state. Later in this section, we discuss all cases in which node identifiers should be added to the queue.

ReCircle uses a method called MLOOKUP(id) for fixing a possible problematic area id on the identifier space. MLOOKUP(id) does the following. First, it performs a greedy routing, similar to a Chord lookup, to the problem area defined by the identifier id. Once it routes to id, it fixes the geometry there by triggering the same afore-mentioned mechanism that is periodically carried out every δ time units. The MLOOKUP then continues to fix the ring in the clockwise direction (Figure 2). Second, an MLOOKUP spreads the fixing process by generating new MLOOKUPs for random identifiers on the ring; hence triggering the fixing mechanism at random places on the identifier. This is accomplished by enqueuing id into random nodes from the nodes routing table (lines 23–24). This is shown in Figure 1. Third, as an optimization, an MLOOKUP attempts to optimistically fix any wrong successor and predecessor pointers while routing by calling the UPDATE method (line 30).

Periodically, after every γ time units (line 14), each node tries to fix the geometry of the overlay by generating MLOOKUPs to identifiers in its queue. Furthermore, whenever p makes an MLOOKUP(q), then q also makes an MLOOKUP(p).

ReCircle provides knobs to tradeoff bandwidth consumption and the time taken to converge to a ring geometry. This tradeoff can be achieved by controlling the amount and rate of spreading the fixing procedure. The number of times the fixing procedure is spread is equivalent to the number of new MLOOKUPs generated. As mentioned earlier, new MLOOKUPs are generated while routing an MLOOKUP (lines 23–24). Here, we employ a *fanout* parameter f that controls how many new MLOOKUPs are generated (line 21). Higher values of the fanout will result in more concurrent MLOOKUPs, hence consuming more bandwidth but converging in lesser time. Similarly, the rate of spreading the fixing procedure is equivalent to the rate at which MLOOKUPs are started. Since new MLOOKUPs are started periodically by dequeuing, this rate can be controlled via the time period γ, and the number of MLOOKUPs generated in each period, denoted as MLKUPS_PER_PERIOD (line 15).

A. Merging multiple overlays

The merger of multiple overlays might be required in two cases. First, an existing overlay can split into multiple overlays due to an underlying network partition. When the network partition ceases, the overlays should merge back into a single overlay. Second, it could happen that multiple overlays are created independently of each other, and later, their administrators decide to merge them due to overlapping interests or change in policy.

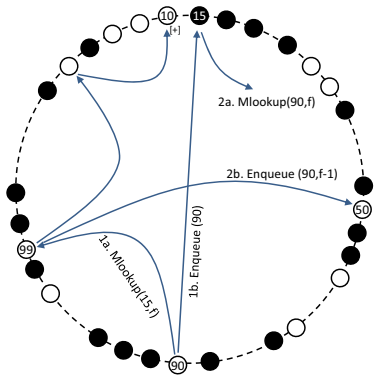

Figure 1. White nodes belong to an overlay $O1$, while black nodes belong to another overlay, $O2$. The merger starts when 15 is added to the queue of 90. (1a) 90 makes an MLOOKUP(15) to fix the ring geometry around identifier 15, and also asks 15 (1b) to make an MLOOKUP for 90, which will result in fixing the ring around 90 (2a). While routing the MLOOKUP(15), 99 shares the merger information with a random node, 50, from its routing table (2b). This will eventually result in an MLOOKUP(50) from 15 that will fix the ring around 50 (not shown in figure). Details of MLOOKUP(15) ending at 10, denoted as [+], are shown in Figure 2.

Passive lists [24], maintained by each node, can be used to detect underlying network partitions and mergers. Whenever a node detects another node n as failed, it adds n to its passive list. Hence, a network partition will result in nodes from one partition being added to passive lists of nodes in the other partition. Each node periodically pings nodes in its passive list to check if a failed node is alive again. When this occurs, it implies that an earlier network partition has ceased and the underlying network has merged. At this point, the overlay merger process can be started. If the network stays partitioned for a long duration, passive lists can become obsolete. In such cases, an administrator has to trigger the overlay merger mechanism when the underlying network merges.

Two independent overlays can be merged into a single overlay using Algorithm 1. The merger can be triggered via connecting the overlays by adding the identifier of any node from one overlay to the queue of any node from the other overlay[1] either by the passive lists mechanism, or an administrator. An MLOOKUP will be generated for the node in the queue. As noted earlier, an MLOOKUP(m) first routes to the problematic area m, terminating at a node n such that $m \in [n, n.succ]$. Then, the geometry is fixed by setting $n.succ := m$, and the two overlays are merged on

[1]The higher the number of connections between the two overlays, the faster the overlay will converge.

Algorithm 1 ReCircle

```
1: every δ time units at p
2:     sendto succ : GETPRED(succ)
3: end event

4: receipt of GETPRED(psucc) from m at n
5:     sendto m : GETPREDRES(pred, sl)
6:     if psucc ≠ n then
7:         queue.enqueue(⟨psucc, f⟩)
8:     UPDATE(m)
9: end event

10: receipt of GETPREDRES(succp, succsl) from m at n
11:     UPDATE(succp)
12:     UPDATESUCCESSORLIST(succsl)
13: end event

14: every γ time units and queue ≠ ∅ at p
15:     for i ← 1:MLKUPS_PER_PERIOD and queue ≠ ∅ do
16:         ⟨q, F⟩ := queue.dequeue()
17:         sendto p : MLOOKUP(q, F)
18:         sendto q : MLOOKUP(p, F)
19: end event

20: receipt of MLOOKUP(id, F) from m at n
21:     if F > 1 then
22:         F := F − 1
23:         r := randomnodeinRT()
24:         at r : queue.enqueue(⟨id, F⟩)
25:     if id ≠ n and id ≠ succ then
26:         if id ∈ (n, succ) then
27:             sendto id : GETPRED(succ)
28:         else
29:             sendto closestpreceding(id) : MLOOKUP(id, F)
30:     UPDATE(id)
31: end procedure

32: procedure UPDATE(candidate) at n
33:     if candidate ∈ (n, succ) then
34:         succ := candidate
35:     else if pred = nil or candidate ∈ (pred, n) then
36:         pred := candidate
37: end procedure
```

the identifier space around identifier m. The merger process is continued by issuing new MLOOKUPs. Consider Figure 2, where $n = 10$ and $m = 15$. An MLOOKUP, to propagate the merger, is needed between 10 and $10.succ = 30$ because a merger between overlays $O1$ and $O2$ can result in several nodes from $O2$ to be placed between a node 10 in $O1$ and 10's successor. 10 accomplishes this propagation by asking 15 (step 2a) to enqueue 30 (line 7) as it represents a problematic area. Such a mechanism enables ReCircle to continue merging the overlays clock-wise. Furthermore, as new MLOOKUPs are generated for random identifiers while routing an MLOOKUP, the overlays concurrently merge clock-wise starting at random positions in the identifier space and eventually, converge into one overlay (Fig. 1).

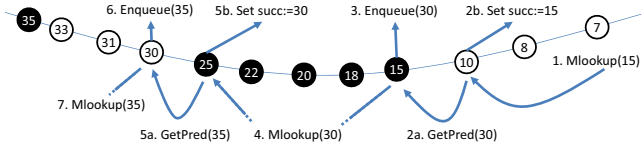

Figure 2. White nodes belong to an overlay $O1$, while black nodes belong to another overlay, $O2$. The figure depicts how new MLOOKUPs are generated when an MLOOKUP(15) terminates at 10; where $10 \in O1$ and $15 \in O2$. Here, the new MLOOKUPs enable the algorithm to continue merging the ring clock-wise.

Note that ideally, the new MLOOKUPs are generated such that the source and destination nodes belong to different overlays.

B. Bootstrapping

The ideas for merging two overlays apply to merging more than two overlays as well; an extreme case of which is bootstrapping where each node can be considered an overlay in itself. Bootstrapping is achieved by creating a structured overlay from a random connected overlay, where each node has some random nodes as neighbours. In our algorithm, each node can be considered an independent structured overlay of size one by pointing to itself as its successor and predecessor. To start bootstrapping, each node adds its neighbours to its *queue*. The algorithm then triggers the merger mechanism by generating MLOOKUPs to nodes in the queue, resulting in a single converged overlay.

C. Termination

An important requirement for a unified algorithm is that under normal scenarios (i.e. no churn), the maintenance cost should be low, for instance, similar to Chord's periodic stabilization. To achieve this, we designed the algorithm to be reactive such that it starts generating more messages than the periodic maintenance mechanism to handle rare events such as bootstrapping or network partitions and mergers. Once such events are catered and the overlay converges, the algorithm stops sending extra messages and the number of messages drops to the only periodic maintenance messages. This property is achieved by not generating new MLOOKUPs when the possible problematic area is already fixed (lines 25 and 6). When the overlay is converged, there will not be any problematic areas and hence, the queues on all nodes will eventually be emptied and no new MLOOKUPs will be started.

IV. EVALUATION

In this section, we evaluate ReCircle by both simulations and experiments on Planetlab[2]. We implemented the algorithm in Kompics [5] and for simulations, we used the King latencies [9] for network delays. The focus of the evaluation is on overlay mergers and bootstrapping as normal scenarios

[2]http://www.planet-lab.org

Parameter		Values
Fanout	f	1 – 5
MLOOKUPs per period	m	1 – 5, ∞
Queue interval	γ	1, 2 (secs)
Periodic maintenance interval	δ	10, 30, 60 (secs)

Table I
RANGE OF PARAMETER VALUES USED FOR SIMULATIONS.

are handled similar to Chord. The two main metrics used are bandwidth consumption and time taken for convergence. The simulations were repeated with 20 different random seeds, and we plot average and 95% confidence intervals in our graphs.

A. Same size networks merge

We first consider the performance of the algorithm when two overlays of same size merge. As the simulation scenario, we created two separate overlays of the same size, and then started the merger algorithm by creating one link between the overlays.

Algorithm 1 uses four parameters:

- *f:* The fanout f used to control the spread of MLOOKUPs (line 7).
- *m:* The number of MLOOKUPs generated in each period γ, shown as MLKUPS_PER_PERIOD on line 15.
- γ*:* The interval after which MLOOKUPs are generated to identifiers stored in the queue (line 14).
- δ*:* The interval after which a node performs periodic stabilization (line 1).

To study the affect of all four parameters, we employed a performance-vs-cost model [15] where we used ranges of values for each parameter. The ranges for parameter values we chose for evaluation are shown in Table I. We used higher values of δ, compared to γ, in line with Li's study [15] on comparing range of values of periodic interval for maintenance in various overlays. In Table I, $m = \infty$ means that the MLOOKUPs are not queued but are instead generated instantly. Each combination of the parameter values was simulated for 20 different random number generation seeds. For each simulation, the combination of parameters had some cost and performance associated with it. For our work, the cost of the algorithm is the *bandwidth* used per peer during the merge process, and the performance is the *time* taken by the algorithm to converge the overlays into one overlay.

Figure 3 shows the results of various combinations of the parameters for a total network size of 2048. Each dot in the graph represents the result of a single experiment for a parameter combination. As is evident from the figure, when the cost is more (higher bandwidth), the performance is better (lower time to convergence). Similarly, less cost (lower bandwidth) results in lower performance (high convergence time). Furthermore, there is a point after which more cost does not increase performance. Similarly, there is a limit to the minimum cost.

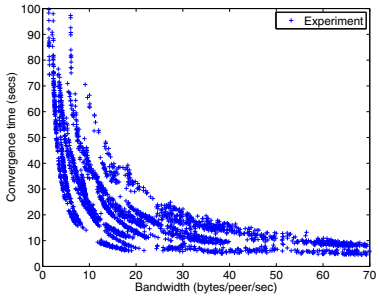

Figure 3. A performance vs cost comparison when two networks, each of size 1024, merge.

Further analysis (omitted due to space constraints) of the performance-vs-cost experiment (Fig 3) shows that δ does not influence the results much. Similarly, increasing γ from 1 second to 2 seconds does not help much either. Hence for the next evaluations, we use $\delta = 60$ second and $\gamma = 1$ seconds.

Affect of Fanout (f) and MLookups per period (m): Next, we discuss the affect of f and m on the cost and performance of the algorithm. Figure 4 shows the convergence time, while Figure 5 shows the bandwidth consumption, for different values of f and m. For $f = 1$, the convergence time is high, yet ReCircle consumes minimum bandwidth. This is an expected behaviour as concurrent MLOOKUPs are not generated when $f = 1$ at line 23 and the merge process continues linearly. Similarly, as we increase the value of f, the convergence time drops slower, while the bandwidth increases exponentially. This trend applies to all simulated values of m, which implies that after a certain value of f, increasing f will only increase cost without significant improvement in performance.

Figure 6 and 7 plot the performance and cost respectively for various values of m. The bandwidth consumption increases logarithmically with m, while time to convergence drops slowly.

An important aspect of the algorithm is that in case there is no churn, ReCircle only sends the periodic maintenance messages. As soon as a rare event that results in churn occurs, such as merger of multiple overlays, the algorithm reacts to it by consuming more bandwidth. Once the overlay converges, the overhead messages die out and the bandwidth consumption drops back. This is shown in Figure 8, where two overlays are merged after 10 seconds. The Y-axis denotes the bandwidth consumed per peer in every 200 milliseconds. As evident from the figure, bandwidth consumption increases to merge the overlays. Once the overlays converge into a single overlay, the bandwidth consumption reduces to the level of before the merger.

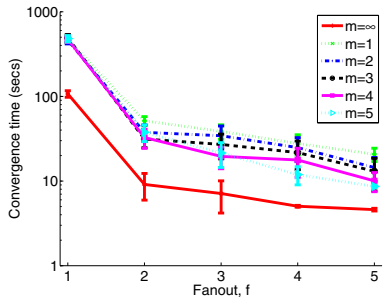

Figure 4. Convergence time for various values of f, where $n = 2048$, $delta = 60$ secs, and $\gamma = 1$ sec.

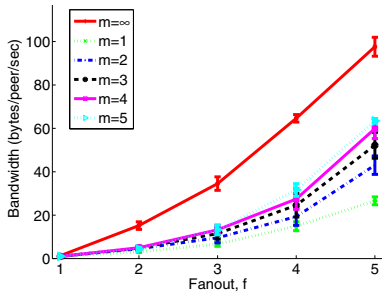

Figure 5. Bandwidth consumption for various values of f, where $n = 2048$, $delta = 60$ secs, and $\gamma = 1$ sec.

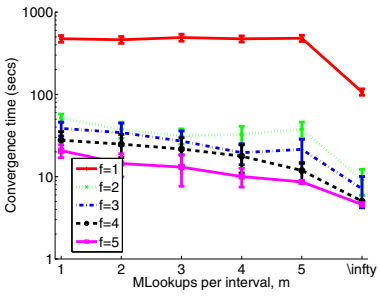

Figure 6. Convergence time for various values of m.

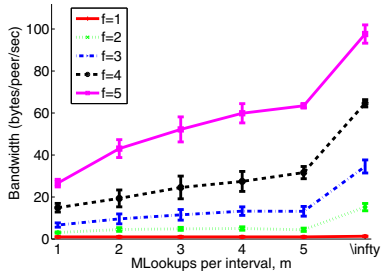

Figure 7. Bandwidth consumption for various values of m.

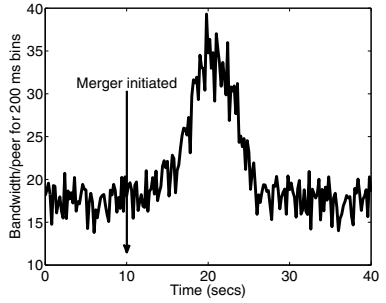

Figure 8. Bandwidth consumption for 200 milliseconds bins, showing termination of reactive messages after convergence.

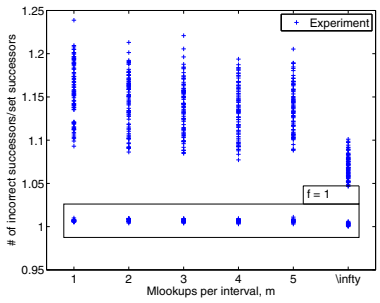

Figure 9. Ratio, during the merger process, of the number of times successor is set versus the number of incorrect successors when the merger started.

B. Set successor calls during merger

Distributed applications build on top of a structured overlay network assign responsibilities to participating nodes based on the region of the identifier space between a node, and its successor and predecessor in the overlay. A change in the successor or predecessor pointers of a node n re-assigns responsibilities between nodes in n's vicinity, which requires action on behalf of the application. For instance, in Distributed Hashtables (DHTs) built on overlays e.g. Cassandra, a node is responsible for storing all data items with keys between its identifier and its immediate neighbour's identifiers. Here, whenever a successor pointer changes, responsibilities are re-defined and data has to be transferred from one node to another. Hence, it is desirable to have a minimum number of unnecessary calls to set the successor of a node, for instance, during merging multiple overlays to avoid unneeded data transfers. In this experiment, we merged two overlays and measured the number of set successor calls, s, and compared it to the number of incorrect successors w at the point the overlays started to merge. Ideally, s should be equal to w, but is difficult to achieve due of decentralization. Figure 9 shows the ratio $\frac{s}{w}$ for a range of values of f (1–5) and m (1–5, and ∞). The graph shows that $f = 1$ has the minimum ratio, and hence would result in minimum data transfer. This shows that if an overlay stores huge data items under keys, the overall time (time for correcting routing pointers and moving data items to new responsible nodes) for $f = 1$ might be lesser than for larger values of f. In the light of this experiment, when higher values of f are used, instead of immediately transferring data when the responsibility of a node changes, a periodic or delayed data exchange mechanism should be used to transfer data among nodes. Using such a technique will avoid transferring data unnecessarily when the merger is under progress.

C. Various network size

Next, we studied the effect of f on the algorithm for different network sizes, while using $m = \infty$. Figure 10 and 11 show the convergence time and bandwidth consumption for various network sizes, depicting that the trend remains the same. Furthermore, for the same value of f, bandwidth consumption per peer is lower, and convergence time higher, for large network sizes compared to smaller sizes. This owes to the fact that information in gossip algorithms spreads logarithmically to the system size. Since f is used to limit the depth of the gossip, the same value of f spreads information at a lower rate in a large network compared to a smaller network. Hence, for a given value of f, the cost

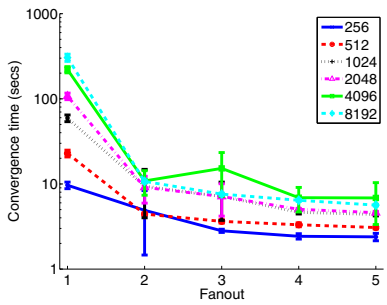

Figure 10. Convergence time for various network sizes, where $m = \infty$, $t = 60$ secs, and $r = 1$ sec.

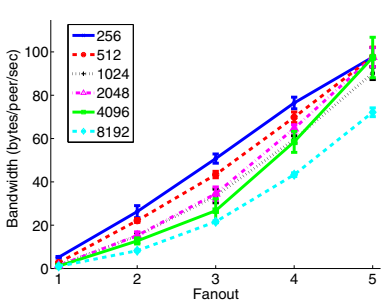

Figure 11. Bandwidth consumption for various network sizes, where $m = \infty$, $t = 60$ secs, and $r = 1$ sec.

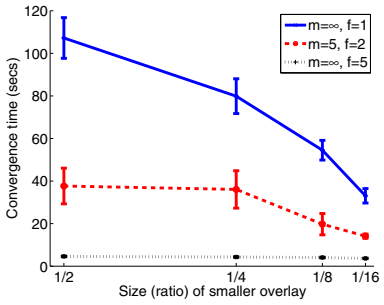

Figure 12. Convergence time when overlays of different size merge.

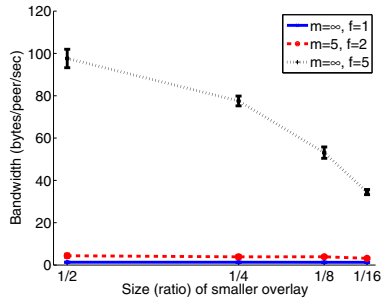

Figure 13. Bandwidth consumption when overlays of different size merge.

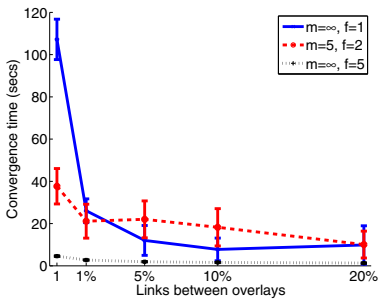

Figure 14. Convergence time when multiple links trigger the merge process.

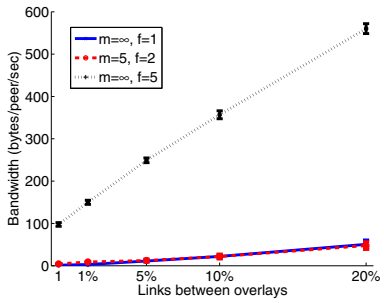

Figure 15. Bandwidth consumption when multiple links trigger the merge process.

and performance are lower for larger networks than smaller networks. This implies that for large networks, higher values of f can be used.

D. Networks of different size merge

In this section, we evaluate the cost and performance of the algorithm when overlays of different sizes merge, which is a common scenario as network partitions are usually of unequal sizes. We expect that the cost should be proportional, and the performance should be inversely proportional, to the size of the smaller network. The reason is that at the start of the merger, when using uniformly random identifiers, the number of wrong successor pointers depends on the size of the smaller network. For instance, for a total network size of 100 nodes, the cost of merging two overlays of sizes 80 and 20 should be lesser than merging overlays of sizes 60 and 40. In our simulations, we created two overlays of different sizes, and then started the merger by creating a single link between the overlays. Figure 12 and 13 show the results for a total network size of 2048. The x-axis depicts the ratio of the smaller network out of the total network size. We plot results for three combinations of m and f: $m = 2, f = 1$ (high convergence time, low bandwidth consumption), $m = 4, f = 3$ (medium convergence time and bandwidth consumption), and $m = \infty, f = 5$ (low convergence time, high bandwidth consumption). The results confirm that when a smaller network merges into a larger network, the algorithm consumes resources relative to the smaller network. Hence, as the size of the smaller network decreases, the algorithm requires lesser time for convergence and bandwidth.

E. Multiple links start merger

Next, we evaluate a scenario where the merger between two overlays is triggered by creating multiple links between the two overlays instead of a single link. This can happen when multiple nodes detect a network partition and merger. In such a scenario, the merger will be started simultaneously at multiple positions on the identifier space. Intuitively, for higher number of inter-overlay links, the overlays should converge faster while consuming higher bandwidth because the algorithm reacts to the merger concurrently for all the inter-overlay links. This was confirmed in our simulations, as shown in Figures 14 and 15. The x-axis represents the number of links created between the two overlays for triggering the merger. The percentage on the x-axis is out of the total network size of 2048. The figures depict that, while multiple links can reduce the time to convergence, it results in higher bandwidth consumption. Higher percentages of links make $f = 1$ behave like $f > 1$ since the merger happens concurrently at different areas on the identifier space even for $f = 1$.

Parameter	Values
Omega, ω	1 – 5
Message size	10, 20, 30, 40, 50
Gossip time period	0.2, 0.5, 1, 1.5, 2 (secs)
Storage size	2048

Table II
RANGE OF PARAMETER VALUES USED FOR T-MAN [12].

F. Bootstrapping

As discussed in Section III-B, Algorithm 1 can be used for bootstrapping an overlay by considering each node as an overlay of size one and connecting the nodes randomly. In this section, we evaluate the performance of our solution for bootstrapping an overlay of size 2048. We create a random Erdős-Rényi graph $G(n, p)$, where $n = 2048$ and $p = \frac{\ln(n)}{n}$, and ensure that the graph is connected. The graph dictates the layout of the initial overlay that has to be bootstrapped in a converged structured overlay. Each node sets itself as its successor and predecessor, and the bootstrapping process is triggered by making each node add its neighbours to its queue.

We compare our solution to T-Man [13], a well-known gossip-based approach for creating arbitrary structured overlays from a random graph. In T-Man, the last few pointers take time to converge [12], hence, we measure statistics until 99% of the successor pointers are converged. To perform an extensive comparison for bootstrapping between T-Man and our solution, and not to depend on parameter tuning, we again employ a performance-vs-cost model. We use a range of parameter values and repeat each experiment for different seeds. For T-Man, we use the values specified in Table II. The results are plotted in Figure 16, which shows that for the same cost (bandwidth consumption), both algorithms have similar performance in terms of convergence time. A disadvantage of using a specialized bootstrapping algorithm, such as T-Man, is that it requires handing off the bootstrapped overlay to the maintenance protocol which is non-trivial. In comparison, ReCircle does not require such a hand off as it embeds the overlay maintenance logic as well. We discuss such differences and benefits of using ReCircle in detail in Section V.

G. PlanetLab

Next, we evaluated the solution for merging multiple overlays and bootstrapping on a real environment by running experiments on PlanetLab. Due to limited number of physical machines available on PlanetLab, we ran 5 nodes on each machine. We used a single server to gather statistics about how many nodes have a correct successor pointer. Whenever a node updated its successor, it sent a message to the statistics server with the new value of its successor. We compute the fraction of correct successor pointers overtime on the statistics server as it has the identifiers of all the

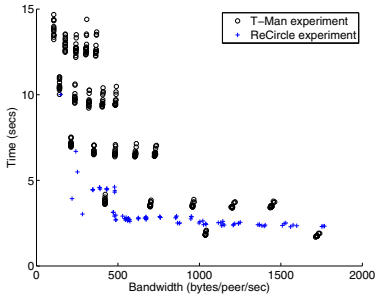

Figure 16. A comparison with T-Man [12] for creating a ring-structured overlay from a random Erdős-Rényi graph for a network size of 2048

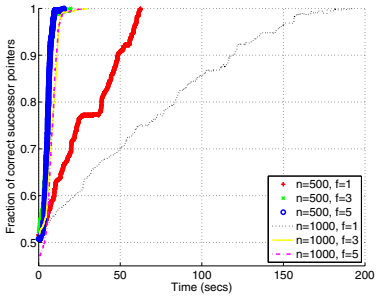

Figure 17. Evaluation on PlanetLab for merging two overlays.

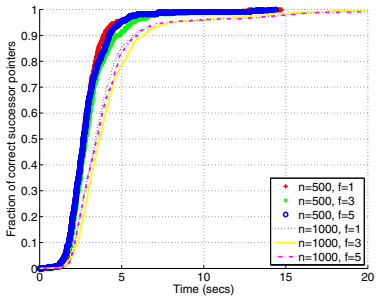

Figure 18. Evaluation on PlanetLab for bootstrapping an overlay.

nodes in the system and the values of each node's current successor.

In the first set of experiments, we evaluated the performance of merging two equal-sized overlays. We created two independent overlays and triggered the merger process by creating a link between the overlays. This was done by adding a random node from one overlay to the queue of a random node in the other overlay. The results for network sizes of 500 and 1000 are shown in Figure 17. Analogous to the simulation results, the convergence rate for $f = 1$ is slower compared to higher values of fanout. Furthermore, with $f = 3$ and 5, most of the successor pointers converge within 10 to 15 seconds.

Finally, we evaluated the performance of bootstrapping an overlay on PlanetLab. We started the nodes as single node overlays by making each node point to itself as its successor

and predecessor. As in the simulations, in this experiment we created an Erdős-Rényi graph at the statistics server using the identifiers of the nodes. Then, the statistics server initiated the bootstrapping process by sending messages to all nodes containing their neighbours as dictated by the generated graph. On receiving such a message, each node added the neighbours to their queue, which triggered the bootstrapping mechanism. Figure 18 depicts the results for this experiment, which shows that an overlay of size 1000 can be created within 10 to 15 seconds. Here, f does not effect the bootstrapping performance much. In ReCircle, f is used to control the spread of the merger information. Since all nodes are already participating in the merger process, $f = 1$ performs the same as higher values of f. Furthermore, the number of neighbours each node has in the graph model used is very small. Hence, higher values of f end up sending redundant messages.

V. Related work

A variety of overlay maintenance algorithms have been proposed over the years [26], [10], [15], and much work has been done to show their resilience to handle churn [17], [18]. These systems can cope with massive failures, thus being able to cope with network partitions as long as a node doesn't loose all its successor-list. Yet, these systems are not intrinsically designed for fast bootstrapping, and cannot merge multiple overlays. In this paper, we show an overlay algorithm that can deal with such extreme events, while being able to perform periodic maintenance like any overlay algorithm. We believe that the underlying principles can be used in other overlays as well.

Bootstrapping a structured overlay is done by constructing a geometry, such as a ring in Chord, from a randomly connected overlay. Shaker *et al.* [25] have presented an algorithm, called Ring Network (RN), for nodes in arbitrary state to converge into a ring topology. While their algorithm can be used for overlay maintenance as well, it cannot converge from certain scenarios [24]. Furthermore, since their algorithm is not reactive to extreme events, it suffers from the same problems as other overlays where the time for convergence when two overlays merge is huge [24].

Montressor *et al.* show how any topology [12], such as a ring [20], can be created from a randomly connected overlay using a gossip-based protocol. However, in their algorithm, it is difficult to detect when the overlay has converged due to decentralization, and thus it depends on heuristics to detect termination. Hence, we believe that further investigation is required to study how these algorithms can be synchronized such that once the topology is built, it can be handed over to the overlay maintenance protocol. On the other hand, ReCircle does not require any such handover.

Recent work has identified the need for structured overlays to handle network partitions and mergers [8], [23], [7]. Datta *et al.* [8], [7] show how to merge multiple P-Grid [4] overlays. P-Grid is a tree-based overlay, in contrast, we focus on ring-based overlays. Shafaat *et al.* [23], [24] and Kis *et al.* [14] present terminating algorithms for merging multiple ring-based overlays. All of these algorithms are triggered for performing the merger, and then terminate after convergence, thus giving the control back to the overlay maintenance protocols. This can lead to two problems. First, the implications of such a terminating algorithm on the overlay maintenance algorithm is not well studied. Coupled with a separate bootstrapping protocol further complicates the interaction between the algorithms. Second, a system developer has to implement and maintain separate mechanisms to address each problem, which can lead to unnecessary complexities.

SkipNet [10] is designed for spanning across multiple organizations and includes mechanisms for recovery when organizations disconnect [11]. These mechanisms rely on the fact that nodes within an organization are places contiguously on the overlay ring. In contrast, most of the overlays place nodes uniformly at random on the identifier space. Hence, SkipNet's mechanisms for partitions cannot be applied to other overlays. ReCircle does not depend on any such requirement; hence, it can be applied to SkipNet, as well as other ring-based overlays. Furthermore, ReCircle provides efficient bootstrapping.

VI. Conclusion

Structured overlay networks are designed for dynamic environments and touted to be scalable, fault-tolerant and self-organizing. Therefore, apart from dealing with normal churn rates, we argue that they should intrinsically be able to handle rare but extreme events such as bootstraping, flash crowds, and network partitions and mergers. We have presented ReCircle, an overlay algorithm that deals with all such cases. Under normal execution, our algorithm exchanges messages periodically like any other overlay maintenance protocol. On the other hand, we designed ReCircle to be reactive to extreme events so that it can converge faster when such events occur.

We have evaluated ReCircle in detail for various scenarios through simulations and experiments on PlanetLab. Additionally, we have illustrated how ReCircle provides tunable knobs to tradeoff between cost (bandwidth consumption) and performance (time to convergence) while handling extreme scenarios.

Future work: While this paper focuses on routing-level issues, we believe an interesting future direction will be to study data-level issues, such as consistency and availability while an overlay is bootstrapping or going through partitions and mergers.

References

[1] "Egypt's big Internet disconnect, 2011," Dec. 2011, http://www.guardian.co.uk/commentisfree/2011/jan/31/egypt-internet-uncensored-cutoff-disconnect.

[2] "ISP quarrel partitions Internet, 2008," Dec. 2011, http://www.wired.com/threatlevel/2008/03/isp-quarrel-par/.

[3] "Taiwan earthquake shakes Internet, undersea cable damage, 2006," Dec. 2011, http://www.theregister.co.uk/2006/12/27/boxing_day_earthquake_taiwan/.

[4] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt, "P-Grid: a self-organizing structured P2P system," *SIGMOD Record*, vol. 32, no. 3, pp. 29–33, 2003.

[5] C. Arad, J. Dowling, and S. Haridi, "Developing, simulating, and deploying peer-to-peer systems using the kompics component model," in *Proceedings of the 4th International Conference on Communication System Software and Middleware (COMSWARE 2009)*, Dublin, Ireland, June 2009.

[6] F. J. C. Labovitz, A. Ahuja, "Experimental Study of Internet Stability and Wide-Area Backbone Failures," University of Michigan, Tech. Rep. CSE-TR-382-98, November 1998.

[7] A. Datta, "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)*. Boston, MA, USA: IEEE Computer Society, July 2007, pp. 109–118.

[8] A. Datta and K. Aberer, "The Challenges of Merging Two Similar Structured Overlays: A Tale of Two Networks," in *Proceedings of the 1st International Workshop on Self-Organizing Systems (IWSOS'06)*, ser. Lecture Notes in Computer Science (LNCS), vol. 4124. Springer-Verlag, 2006, pp. 7–22.

[9] K. P. Gummadi, S. Saroiu, and S. D. Gribble, "King: estimating latency between arbitrary internet end hosts," in *IMW '02: Proc. of the 2nd ACM SIGCOMM Workshop on Internet measurment*. New York, NY, USA: ACM, 2002, pp. 5–18.

[10] 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: USENIX, Mar. 2003.

[11] N. J. A. Harvey, M. B. Jones, M. Theimer, and A. Wolman, "Efficient recovery from organizational disconnects in skipnet," in *IPTPS*, 2003, pp. 183–196.

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

[13] M. Jelasity, A. Montresor, and Ö. Babaoglu, "T-Man: Gossip-based fast overlay topology construction," *Computer Networks*, vol. 53, no. 13, pp. 2321–2339, 2009.

[14] Z. Kis and R. Szabo, "Chord-zip: a chord-ring merger algorithm," *Communications Letters, IEEE*, vol. 12, no. 8, pp. 605 –607, Aug. 2008.

[15] J. Li, "Routing tradeoffs in dynamic peertopeer networks," PhD Dissertation, MIT—Massachusetts Institute of Technology, Massachusetts, USA, Nov. 2005.

[16] D. Liben-Nowell, H. Balakrishnan, and D. R. Karger, "Analysis of the Evolution of Peer-to-Peer Systems," in *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC'02)*. New York, NY, USA: ACM Press, 2002, pp. 233–242.

[17] D. Liben-Nowell, H. Balakrishnan, and D. R. Karger, "Observations on the Dynamic Evolution of Peer-to-Peer Networks," in *Proceedings of the 1st Intl. Workshop on Peer-to-Peer Systems (IPTPS'02)*, ser. LNCS, vol. 2429. Springer-Verlag, 2002.

[18] R. Mahajan, M. Castro, and A. Rowstron, "Controlling the Cost of Reliability in Peer-to-Peer Overlays," in *Proceedings of the 2nd Intl. Workshop on Peer-to-Peer Systems (IPTPS'03)*, ser. LNCS, vol. 2735. Springer-Verlag, 2003, pp. 21–32.

[19] A. Mislove, A. Post, A. Haeberlen, and P. Druschel, "Experiences in building and operating ePOST, a reliable peer-to-peer application," in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*, W. Zwaenepoel, Ed. ACM European Chapter, April 2006.

[20] 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, Aug. 2005.

[21] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "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*. Berkeley, CA, USA: USENIX Association, 2003, pp. 1–1.

[22] V. Paxson, "End-to-end routing behavior in the Internet," *IEEE/ACM Transactions on Networking (TON)*, vol. 5, no. 5, pp. 601–615, 1997.

[23] T. M. Shafaat, A. Ghodsi, and S. Haridi, "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, Sep. 2007, pp. 132–139.

[24] T. M. Shafaat, A. Ghodsi, and S. Haridi", "Dealing with network partitions in structured overlay networks," *Peer-to-Peer Networking and Applications (PPNA)*, vol. 2, no. 4, pp. 334–347, 2009.

[25] 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)*. IEEE Computer Society, Aug. 2005, pp. 39–46.

[26] 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)*, vol. 11, no. 1, pp. 17–32, 2003.