A Practical Approach to Network Size Estimation for Structured Overlays

Tallat M. Shafaat[1], Ali Ghodsi[2], and Seif Haridi[1]

[1] Royal Institute of Technology (KTH),
School of Information and Communication, Stockholm, Sweden
{tallat,haridi}@kth.se
[2] Computer Systems Laboratory,
Swedish Institute of Computer Science, Stockholm, Sweden
ali@sics.se

Abstract. Structured overlay networks have recently received much attention due to their self-* properties under dynamic and decentralized settings. The number of nodes in an overlay fluctuates all the time due to churn. Since knowledge of the size of the overlay is a core requirement for many systems, estimating the size in a decentralized manner is a challenge taken up by recent research activities. Gossip-based Aggregation has been shown to give accurate estimates for the network size, but previous work done is highly sensitive to node failures. In this paper, we present a gossip-based aggregation-style network size estimation algorithm. We discuss shortcomings of existing aggregation-based size estimation algorithms, and give a solution that is highly robust to node failures and is adaptive to network delays. We examine our solution in various scenarios to demonstrate its effectiveness.

1 Introduction

Structured peer-to-peer systems such as Chord [20] and Pastry [18] have received much attention by the research community recently. These systems are typically very scalable and the number of nodes in the system immensely varies. The network size is, however, a global variable which is not accessible to individual nodes in the system as they only know a subset of the other nodes. This information is, nevertheless, of great importance to many structured p2p systems, as it can be used to tune the rates at which the topology is maintained. Moreover, it can be used in structured overlays for load-balancing purposes [4], deciding successor-lists size for resilience to churn [12], choosing a level to determine outgoing links [14], and for designing algorithms that adapt their actions depending on the system size [1].

Due to the importance of knowing the network size, several algorithms have been proposed for this purpose. Out of these, gossip-based aggregation algorithms [8], though having higher overhead, provide the best accuracy [17]. Consequently, we focus on gossip-based aggregation algorithms in this paper. While

K.A. Hummel and J.P.G. Sterbenz (Eds.): IWSOS 2008, LNCS 5343, pp. 71–83, 2008.

aggregation algorithms can be used to calculate different aggregates, e.g. average, maximum, minimum, variance etc., our focus is on counting the number of nodes in the system.

Although Aggregation [8] provides accurate estimates, it suffers from a few problems. First, Aggregation is highly sensitive to the *overlay topology* that it is used with. Convergence of the estimate to the real network size is slow for non-random topologies. On the contrary, the majority of structured p2p overlays have non-random topologies. Thus, it is not viable to directly use Aggregation in these systems. Second, Aggregation works in rounds, and the estimate is considered converged after a predefined number of rounds. As we discuss in section 4.1, this can be problematic. Finally, Aggregation is highly sensitive to node failures.

In this paper, we suggest a gossip algorithm based on Aggregation to be executed continously on every node to estimate the total number of nodes in the system. The algorithm is aimed to work on structured overlay networks. Furthermore, the algorithm is robust to failures and adaptive to the network delays in the system.

Outline. Section 2 serves as a background for our work. Section 3 describes our solution and discusses how the proposed solution handles the dynamism in the network. Thereafter, section 4 gives a detailed evaluation of our work. Section 5 discusses the related work, and finally, section 6 concludes.

2 Background

In this section, we briefly define a model of a ring-based structured overlay underlying our algorithm. We also describe the original Aggregation algorithm suggested by Jelasity et. al. [8].

2.1 A Model of a Ring-Based Structured Overlay Network

A ring-based structured overlay network consists of nodes which are assigned unique identifiers belonging to a ring of identifiers $\mathcal{I} = \{0, 1, \cdots, N-1\}$ for some large constant N. This is general enough to encompass many existing structured peer-to-peer systems such as Chord[20], Pastry[18] and many others.

Every node has a pointer to its successor, which is the first node met going clockwise on the ring. Every node also has a pointer to its predecessor, which is first node met going anti-clockwise on the ring. For instance, in a ring of size $N = 1024$ containing the nodes $\mathcal{P} = \{10, 235, 903\}$, we have that $succ_{\mathcal{P}}(10) = 235$, $succ_{\mathcal{P}}(903) = 10$, $pred_{\mathcal{P}}(235) = 10$, and $pred_{\mathcal{P}}(10) = 903$.

In this paper, we assume that there exists an out-of-bound mechanism to make all of the predecessor and successor pointers correct. This can, for example, be achieved by using periodic stabilization[20].

Apart from successor and predecessor pointers, each node has additional long pointers in the ring for efficient routing. Different structured overlays use different schemes to place these extra pointers. Our work is independent of how the extra pointers are placed.

While this model looks specific to ring topologies, other structured topologies use similar metrics, for instance, the XOR-metric [16] or butterfly networks [14] Our work can be extended to incorporate metrics other then that employed in ring-based overlays.

2.2 Gossip-Based Aggregation

The Aggregation algorithm suggested by Jelasity et. al. [8] is based on push-pull gossiping, shown in Algorithm 1.

Algorithm 1. Push-pull gossip executed by node p in Aggregation [8]

1: do periodically every δ time units	do forever
2: q ← getneighbour()	s_q ← receive(*)
3: send s_p to q	send s_p to sender(s_q)
4: s_q ← receive(q)	s_q ← update(s_p, s_q)
5: s_q ← update(s_p, s_q)	
(a) Active thread	(b) Passive thread

The method GetNeighbour returns a uniform random sampled node over the entire set of nodes provided by an underlying sampling service like Newscast [7]. The method Update computes a new local state based on the node p's current local state s_p and the remote node's state s_q.

The time interval δ after which the active thread initiates an exchange is called a *cycle*. Given that all nodes use the same value of δ, each node roughly participates in two exchanges in each cycle, one as an initiator and the other as a receipient of an exchange request. Thus, the total number of exchanges in a cycle are roughly equal to $2 \cdot n$, where n is the network size.

For network size estimation, one random node sets its local state to 1 while all other nodes set their local states to 0. The global average is thus $\frac{1}{n}$, where n is the number of nodes. Executing the aggregation algorithm for a number of cycles decreases the variance of local states of nodes but keeps the global average the same. Thus, after convergence, a node p can estimate the network size as $\frac{1}{s_p}$. For network size estimation, Update(s_p, s_q) returns $\frac{s_p+s_q}{2}$.

Aggregation [8] achieves up-to-date estimates by periodically restarting the protocol, i.e. local values are re-initialized and aggregation starts again. This is done after a predefined number of cycles γ, called an epoch.

The main disadvantage of Aggregation is that a failure of a single node early in an epoch can severely effect the estimate. For example, if the node with local state 1 crashes after executing a single exchange, 50% of the value will disappear, giving $2 \cdot n$ as the final size estimate. This issue is further elaborated in section 4.3. Another disadvantage, as we discuss in section 4.1, is predefining the epoch length γ.

3 The Network Size Estimation Algorithm

A naive approach to estimate the network size in a ring-based overlay would be pass a token around the ring, starting from, say node i and containing a variable v initialized to 1. Each node increments v and forwards the token to its successor i.e. the next node on the ring. When the token reaches back at i, v will contain the network size. While this solution seems simple and efficient, it suffers from multiple problems. First, it is not fault-tolerant as the node with the token may fail. This will require complicated modifications for regenerating the token with the current value of v. Second, the naive approach will be quite slow, as it will take $O(n)$ time to complete. Since peer-to-peer systems are highly dynamic, the actual size may have changed completed by the time the algorithm finishes. Lastly, at the end of the naive approach, the estimate will be known only to node i which will have to broadcast it to all nodes in the system. Our solution aims at solving all these problems at the expense of a higher message complexity than the naive approach.

Our goal is to make an algorithm where each node tries to estimate the average inter-node distance, Δ, on the identifier space, i.e. the average distance between two consecutive nodes on the ring. Given a correct value of Δ, the number of nodes in the system can be estimated as $\frac{N}{\Delta}$, N being the size of the identifier space.

Every node p in the system keeps a local estimate of the average inter-node distance in a local variable d_p. Our goal is to compute $\frac{\sum_{i \in \mathcal{P}} d_i}{|\mathcal{P}|}$. The philosophy underlying our algorithm is the observation that at any time the following invariant should always be satisfied: $N = \sum_{i \in \mathcal{P}} d_i$.

We achieve this by letting each node p initialize its estimate d_p to the distance to its successor on the identifier space. In other words, $d_p = succ(p) \ominus p$, where $\ominus$ represents subtraction modulo N. Note that if the system only contains one node, then $d_p = N$. Clearly, a correctly initialized network satisfies the mentioned invariant as the sum of the estimates is equal to N.

To estimate Δ, we employ a modified aggregation algorithm. Since we do not have access to random nodes, we implement the GetNeighbour method in Alg. 1 by returning a node reached by making a random walk of length h. For instance, to perform an exchange, p sends an exchange request to one of its neighbours, selected randomly, with a hop value h. Upon receiving such a request, a node r decrements h and forwards the request to one of its own neighbours, again selected randomly. This is repeated until h reaches 0, after which the exchange takes place between p and the last node getting the request.

Given that GetNeighbour returns random nodes, after a number of exchanges (logarithmic number of steps, to the network size, as show in [8]), every node will have $d_p = \frac{\sum_{i \in \mathcal{P}} d_i}{|\mathcal{P}|}$. On average in each cycle, each node initiates an exchange once, which takes h hops, and replies to one exchange. Consequently, the number of messages for the aggregation algorithm are roughly $hops \times n + n$ per cycle.

3.1 Handling Churn

The protocol described so far does not take into account the dynamicity of large scale peer-to-peer systems. In this section, we present our solution as an extension of the basic algorithm described in section 3 to handle dynamism in the network.

The basic idea of our solution is that each node keeps different levels of estimates, each with a different accuracy. The lowest level estimate is the same as d_n in the basic algorithm. As the value in the lowest level converges, it is moved to the next level. While this helps by having high accuracy in upper levels, it also gives a continuous access to a correct estimated value while the lowest level is re-initialized. Furthermore, we restart the protocol adaptively, instead at a predefined interval.

Our solution is shown in Algorithm 2. Each node n keeps track of the current epoch in *nEpoch* and stores the estimate in a local variable *ndvalue* instead of d_n in the basic algorithm. *ndvalue* is a tuple of size l, i.e.

$$ndvalue = (ndvalue_{l-1}, ndvalue_{l-2}, \cdots, ndvalue_0)$$

The tuple values are called levels. The value at level 0 is the same as d_n in the basic algorithm and has the most recent updated estimate but with high error, while level $l-1$ has the most accurate estimate but incorporates updates slowly.

A node n initializes its estimate, method InitializeEstimate in Alg. 2, by setting level 0 to $succ_{\mathcal{P}}(n) \ominus n$. The method LeftShiftLevels moves the estimate of each level one level up, e.g. left shifting a tuple $e = (e_{l-1}, e_{l-2}, \cdots, e_0)$ gives $(e_{l-2}, e_{l-3}, \cdots, e_0, nil)$. The method Update(a, b) returns an average of each level, i.e. $(\frac{a_{l-1}+b_{l-1}}{2}, \frac{a_{l-2}+b_{l-2}}{2}, \cdots, \frac{a_0+b_0}{2})$.

To incorporate changes in the network size due to churn, we also restart the algorithm, though not after a predefined number of cycles, but adaptively by analyzing the variance. We let the lowest level converge and then restart. This duration may be larger than a predefined γ or less, depending on the system-wide variance in the system of the value being estimated. We achieve adaptivity by using a sliding window at each node. Each node stores values of the lowest level estimate for each cycle in a sliding window W of length w. If the coefficient of variance of the sliding window is less than a desired accuracy e.g. 10^{-2}, the value is considered converged, denoted by the method Converged in Alg. 2.

Once the value is considered to have converged based on the sliding window, there are different methods of deciding which node will restart the protocol, denoted by the method IAmStarter in Alg. 2. One way is as used in [8], each node restarts the protocol with probability $1/\hat{n}$, where $\hat{n}$ is the estimated network size. Given a reasonable estimate in the previous epoch, this will lead to one node restarting the protocol with high probability. It does not matter if more than one node restarts the protocol in our solution. On the contrary, multiple nodes restarting an epoch in [8] is problematic since only one node should set its local estimate to 1 in an epoch. Consequently, an epoch has to be marked with a unique identifier in [8]. Another method is that a node n restarts the protocol which has $0 \in [n, n.succ)$. For our simulations, we use the first method.

Algorithm 2. Network size estimation

```
1: every δ time units at n
2:     if converged() and iamstarter() then
3:         simpleBroadcast(nEpoch)
4:     end if
5:     sendto randomNeighbour() : ReqExchange(hops, nEpoch, ndvalue)
6: end event

7: receipt of ReqExchange(hops, mEpoch, mdvalue) from m at n
8:     if hops > 1 then
9:         hops := hops − 1
10:        sendto randomNeighbour() : ReqExchange(hops, mEpoch, mdvalue)
11:    else
12:        if nEpoch > mEpoch then
13:            sendto m : ResExchange(false, nEpoch, ndvalue)
14:        else
15:            trigger ⟨ MoveToNewEpoch | mEpoch ⟩
16:            ndvalue := update(ndvalue, mdvalue)
17:            updateSlidingWindow(ndvalue)
18:            sendto m : ResExchange(true, nEpoch, ndvalue)
19:        end if
20:    end if
21: end event

22: receipt of ResExchange(updated, mEpoch, mdvalues) from m at n
23:    if updated = false then
24:        trigger ⟨ MoveToNewEpoch | mEpoch ⟩
25:    else
26:        if nEpoch = mEpoch then
27:            dvalue := mdvalues
28:        end if
29:    end if
30: end event

31: receipt of DeliverSimpleBroadcast(mEpoch) from m at n
32:    trigger ⟨ MoveToNewEpoch | mEpoch ⟩
33: end event

34: upon event ⟨ MoveToNewEpoch | epoch ⟩ at n
35:    if nEpoch < mEpoch then
36:        leftShiftLevels()
37:        initializeEstimate()
38:        nEpoch := epoch
39:    end if
40: end event
```

Once a new epoch starts, all nodes should join it quickly. Aggregation [8] achieves this by the logarithmic epidemic spreading property of random networks. Since we do not have access to random nodes, we use a simple broadcast scheme [3] for this purpose, which is both inexpensive ($O(n)$ messages) and fast ($O(\log n)$ steps). The broadcast is best-effort, as even if it fails, the new epoch number is spread through exchanges.

When a new node joins the system, it starts participating in the size estimation protocol when the next epoch starts. This happens either when it receives the broadcast, or its predecessor initializes its estimate. Until then, it keeps forwarding any requests for exchange to a randomly selected neighbour.

Handling churn in our protocol is much simpler and less expensive on bandwidth than other aggregation algorithms. Instead of running multiple epochs as in [8], we rely on the fact that a crash in our system does not effect the end estimate as much as in [8]. This is explored in detail in section 4.3.

4 Evaluation

To evaluate our solution, we implemented the Chord [20] overlay in an event-based simulator [19]. For the first set of experiments, the results are for a network size of 5000 nodes using the King dataset [5] for message latencies. Since we do not have the King dataset for a 5000 node topology, we derive the 5000 node pair-wise latencies from the distance between two random points in a Euclidean space. The mean RTT remains the same as in the King data. This technique is the same as used in [11]. For larger network sizes, the results are for 10^5 nodes using exponentially distributed message latencies with mean 5 simulation time units. For the figures, $\delta = 8 * \textit{mean-com}$ means the cycle length is 8×5.

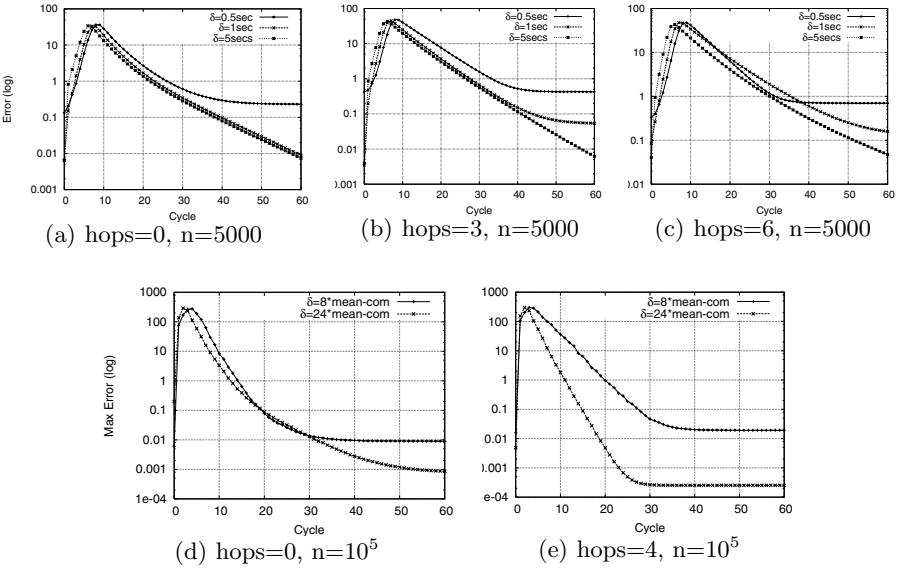
4.1 Epoch Length γ

We investigated the effect of δ on convergence of the algorithm. The results are shown in Figure 1, where error$= \frac{1}{n}\sum_{i=1}^{n} |d_i - \frac{N}{n}|$. It shows that when the ratio between communication delay and δ is significant, e.g. $\delta = 0.5$ *secs* or $8 * \textit{mean-com}$, the aggregate converges slowly and to a value with higher error. For cases where the ratio is insignificant, e.g. $\delta = 5$ *secs* or $24 * \textit{mean-com}$, the convergence is faster and the converged value has lower error. The reason for this behaviour is that when δ is small, the expected number of exchanges per cycle do not occur.

Since δ and γ effect convergence rate and accuracy, our solution aims at having adaptive epoch lengths. Another benefit of using an adaptive approach as ours is that the protocol may converge much before a predefined γ, thus sending messages in vain. If the protocol was restarted, these extra cycles could have been used to get updated aggregate value or include churn effects faster.

4.2 Effect of the Number of Hops

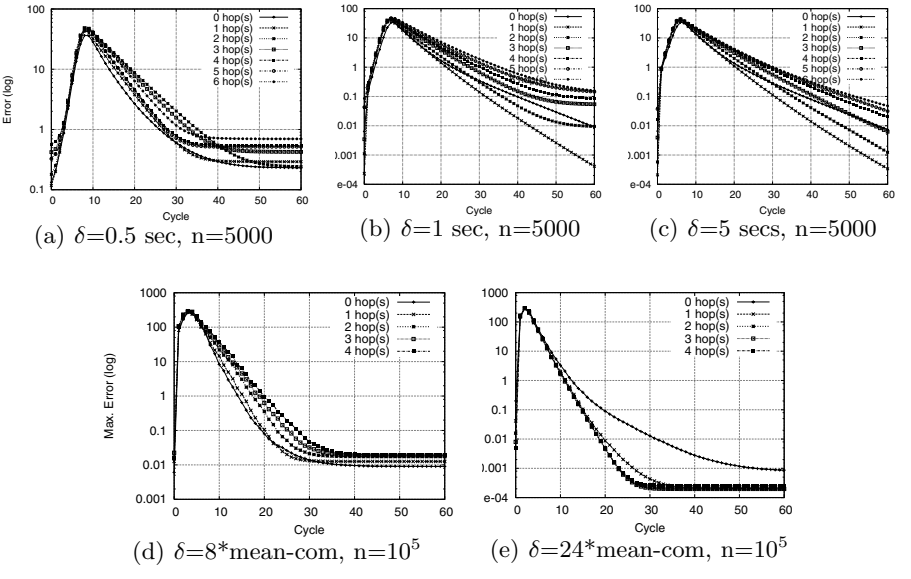
Figure 2 shows convergence of the algorithm for different values of δ and number of hops h to get a random node. For small values of δ, e.g. 0.5 secs and $8 *$



(a) hops=0, n=5000 (b) hops=3, n=5000 (c) hops=6, n=5000

(d) hops=0, n=10^5 (e) hops=4, n=10^5

Fig. 1. Error for the estimate of inter-node distance d in the system

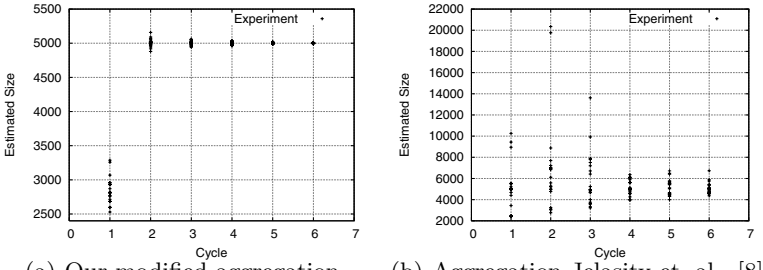


(a) δ=0.5 sec, n=5000 (b) δ=1 sec, n=5000 (c) δ=5 secs, n=5000

(d) δ=8*mean-com, n=10^5 (e) δ=24*mean-com, n=10^5

Fig. 2. Error for the estimate of inter-node distance d in the system

mean-com, $h = 0$ gives best convergence. The reason for this behaviour is that since δ is very small (thus, is comparative to communication delays), having multiple hops will not have enough exchanges in a cycle. Thus, convergence



(a) Our modified aggregation (b) Aggregation Jelasity et. al. [8]

Fig. 3. Estimated size when 50% of the nodes out of 5000 fail suddenly. X-axis gives the cycle at which the sudden death occured in the epoch.

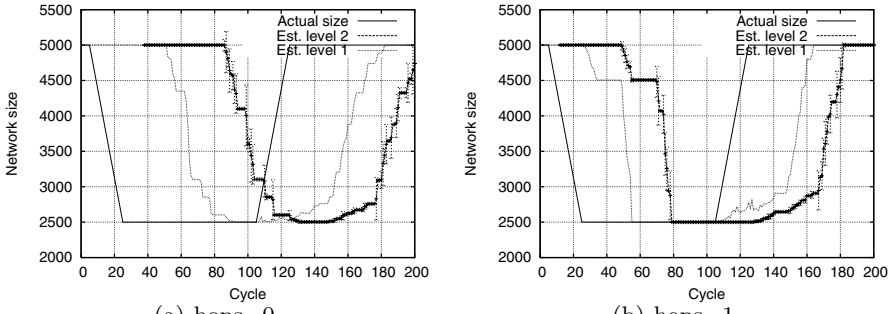
takes longer time and the error is larger for larger values of h. On the contrary, as we increase δ, higher values of h give convergence in lesser time and lower error. These results also advocate to have an adaptive epoch length.

4.3 Churn

Flash Crowds. Next, we evaluated a scenario where a massive node failure occurs. Contrary to [8] where failure of nodes with higher local estimate effects the end estimate more than with lower local estimate, failure of any node is equal in our protocol. The results for a scenario where 50% of the nodes fail at different cycles of an epoch is shown in Figure 3. Our modified aggregation solution, Fig. 3(a), is not as severly affected by the sudden death as the original Aggregation algorithm, fig. 3(b). Infact, in some experiments with Aggregation, the estimate became infinite (not shown in the figure). This happens when all the nodes with non-zero local estimates fail. For our solution, the effect of a sudden death is already negligible if the nodes crash after the third cycle.

Continuous Churn. We ran simulations for a scenario with churn, where nodes join and fail. The results are shown in figure 4, 5 and 7. The results are for extreme churn cases, i.e. 50% nodes fail within a few cycles and afterwards, 50% nodes join within a few cycles. The graphs show how the estimation follows the actual network size. The standard deviation of level 2 is shown as vertical bars, which depicts that all nodes estimate the same size. The standard deviation is high only when a new epoch starts, because while evaluating the mean and standard deviation, some nodes have moved to the new epoch, while others are still in the older epoch. The estimate at level 1 converges to the actual size faster than level 2, but the estimates has higher variance as the standard deviation for level 1 (not shown) is higher than for level 2. Figure 5 also shows that compared to $h = 0$, higher values of h follow the trend of the actual size faster.

Next, we simulated a network of size 4500 and evaluated our algorithm under continous churn. In each cycle, we failed some random nodes and joined new nodes. As explained in section 3, new nodes do not participate in the algorithm



(a) hops=0 (b) hops=1

Fig. 4. Mean estimated size by each node with standard deviation

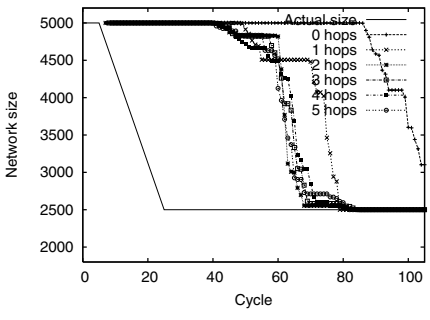


Fig. 5. Mean estimated size for different values of h for level 2. Gives a comparison of how fast the nodes estimation follows the real network size.

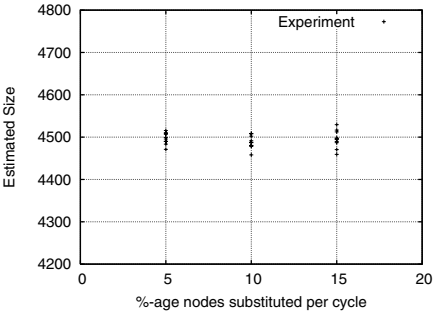


Fig. 6. Estimated network size for 4500 nodes under continuous churn. X-axis gives the percentage of churn events (joins+failures) that occur in each cycle.

till the next epoch starts, yet they can forward requests. Figure 6 shows the results. The plotted dots correspond to the converged mean estimate after 15 cycles for each experiment. The x-axis gives the percentage of churn events, including both failures and joins, that occur in each cycle. Thus, 10% means that $4500 \times \frac{10}{100} \times 15$ churn events occured before the plotted converged value. Figure 6 shows that the algorithm handles continous churn reasonably well.

5 Related Work

Network size estimation in the context of peer-to-peer systems is challenging as these systems are completely decentralized, nodes may fail anytime, the network size varies dynamically over time, and the estimation algorithm needs to continuously update its estimation to reflect the current number of nodes.

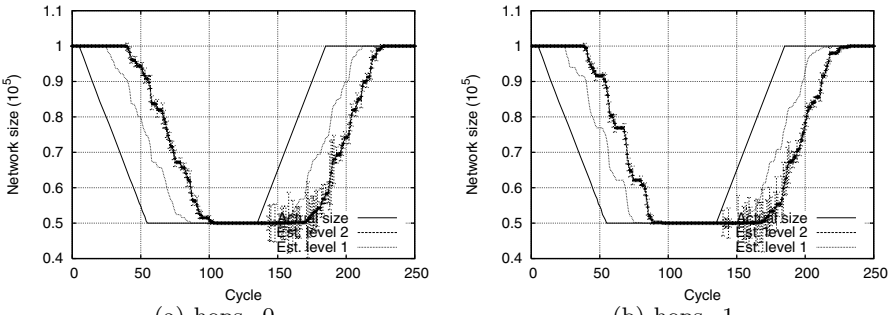


Fig. 7. Mean estimated size by each node with standard deviation

Merrer et. al. [17] compare three existing size estimation algorithms, Sample & Collide [15], Hops Sampling [10] and Aggregation [8], which are representative of three main classes of network size estimation algorithms. Their study yields that although Aggregation is expensive, it produces the most accurate results. Aggregation also has the additional benefit that the estimamte is available on all nodes compared to only at the initiator in the case of Sample & Collide and Hops Sampling. Our work can be seen as an extension of Aggregation, to handle its shortcomings and extend it to non-random topologies, such as structured overlay networks.

The work by Horowitz et. al. [6] is similar to ours in the sense that they also utilize the structure of the system. They use a localized probabilistic technique to estimate the network size by maintaining a structure: a logical ring. Each node estimates the network size locally based on the estimates of its neighbours on the ring. While their technique has less overhead, the estimates are not accurate, the expected accuracy being in the range $n/2 \cdots n$. Their work has been extended by Andreas et. al. [2] specifically for Chord, yet the extended work also suffers similar inaccuracy range for the estimated size. Mahajan et. al. [13] also estimate the network size through the density of node identifiers in Pastry's leafset, yet they neither prove any accuracy range, nor provide any simulation results to show the effectiveness of their technique.

Kempe et. al. [9] have also suggested a gossip-based aggregation scheme, yet their solution focuses only on push-based gossiping. Using push-based gossiping complicates the update and exchange process as a normalization factor needs to be kept track of. On the same, as noted by Jelasity et. al. [8], push-based gossiping suffers from problems when the underlying directed graph used is not strongly connected. Thus, we build our work on push-pull gossip-based aggregation [8]. Similarly, to estimate the network size, Kempe et. al. also propose that one node should initialize its weight to 1, while the other nodes initialize to weight 0, making it highly sensitive to failures early in the algorithm.

The authors of Viceroy [14] and Mercury [1] mention that a nodes distance to its successor can be used to calculate the number of nodes in the system,

but provide no reasoning that the value always converges exactly to the correct value, and thus that their estimate is unbiased.

6 Conclusion

Knowledge of the current network size of a structured p2p system is a prime requirement for many systems, which prompted to finding solutions for size estimation. Previous studies have shown that gossip-based aggregation algorithms, though being expensive, produce accurate estimates of the network size. We have demonstrated the shortcomings in existing aggregation approaches to network size estimation and have presented a solution that overcomes the deficiencies. In this paper, we have argued for an adaptive approach to convergence in gossip-based aggregation algorithms. Our solution is resilient to massive node failures and is aimed to work on non-random topologies such as structured overlay networks.

References

1. Bharambe, A.R., Agrawal, M., Seshan, S.: Mercury: Supporting Scalable Multi-Attribute Range Queries. In: Proceedings of the ACM SIGCOMM 2004 Symposium on Communication, Architecture, and Protocols, OR, USA. ACM Press, New York (2004)
2. Binzenhöfer, A., Staehle, D., Henjes, R.: On the fly estimation of the peer population in a chord-based p2p system. In: 19th International Teletraffic Congress (ITC19), Beijing, China (September 2005)
3. Ghodsi, A.: Distributed k-ary System: Algorithms for Distributed Hash Tables. PhD dissertation, KTH—Royal Institute of Technology, Stockholm, Sweden (December 2006)
4. Godfrey, P.B., Stoica, I.: Heterogeneity and Load Balance in Distributed Hash Tables. In: Proc. of the 24th Annual Joint Conf. of the IEEE Computer and Communications Societies (INFOCOM 2005), FL, USA. IEEE Comp. Society, Los Alamitos (2005)
5. Gummadi, K.P., Saroiu, S., Gribble, S.D.: King: estimating latency between arbitrary internet end hosts. In: IMW 2002: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment, pp. 5–18. ACM, New York (2002)
6. Horowitz, K., Malkhi, D.: Estimating network size from local information. Information Processing Letters 88(5), 237–243 (2003)
7. Jelasity, M., Kowalczyk, W., van Steen, M.: Newscast Computing. Technical Report IR–CS–006, Vrije Universiteit (November 2003)
8. Jelasity, M., Montresor, A., Babaoglu, Ö.: Gossip-based Aggregation in Large Dynamic Networks. ACM Trans. on Computer Systems (TOCS) 23(3) (August 2005)
9. Kempe, D., Dobra, A., Gehrke, J.: Gossip-based computation of aggregate information. In: 44th Symp. on Foundations of Computer Science, FOCS (2003)
10. Kostoulas, D., Psaltoulis, D., Gupta, I., Birman, K., Demers, A.J.: Decentralized schemes for size estimation in large and dynamic groups. In: 4th IEEE International Symp. on Network Computing and Applications (NCA 2005), pp. 41–48 (2005)

11. Li, J., Stribling, J., Morris, R., Kaashoek, M.F.: Bandwidth-efficient management of DHT routing tables. In: Proc. of the 2nd USENIX Symp. on Networked Systems Design and Implementation (NSDI 2005), MA, USA, May 2005, USENIX (2005)
12. Liben-Nowell, D., Balakrishnan, H., Karger, D.R.: Analysis of the Evolution of Peer-to-Peer Systems. In: Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC 2002), pp. 233–242. ACM Press, New York (2002)
13. Mahajan, R., Castro, M., Rowstron, A.: Controlling the Cost of Reliability in Peer-to-Peer Overlays. In: Kaashoek, M.F., Stoica, I. (eds.) IPTPS 2003. LNCS, vol. 2735, pp. 21–32. Springer, Heidelberg (2003)
14. Malkhi, D., Naor, M., Ratajczak, D.: Viceroy: A scalable and dynamic emulation of the butterfly. In: Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC 2002). ACM Press, New York (2002)
15. Massoulié, L., Merrer, E.L., Kermarrec, A., Ganesh, A.J.: Peer counting and sampling in overlay networks: random walk methods. In: Proc. of the 25th Annual ACM Symp. on Principles of Distributed Computing (PODC), pp. 123–132 (2006)
16. Maymounkov, P., Mazieres, D.: Kademlia: A Peer-to-Peer Information System Based on the XOR metric. In: Druschel, P., Kaashoek, M.F., Rowstron, A. (eds.) IPTPS 2002. LNCS, vol. 2429, pp. 53–65. Springer, Heidelberg (2002)
17. Merrer, E.L., Kermarrec, A.-M., Massoulie, L.: Peer to peer size estimation in large and dynamic networks: A comparative study. In: Proc. of the 15th IEEE Symposium on High Performance Distributed Computing, pp. 7–17. IEEE, Los Alamitos (2006)
18. Rowstron, A., Druschel, P.: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In: Guerraoui, R. (ed.) Middleware 2001. LNCS, vol. 2218, pp. 329–350. Springer, Heidelberg (2001)
19. SicsSim (2008), `http://dks.sics.se/iwsos08sizeest/`
20. Stoica, I., Morris, R., Karger, D.R., Kaashoek, M.F., Balakrishnan, H.: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In: Proceedings of the ACM SIGCOMM 2001 Symposium on Communication, Architecture, and Protocols, San Deigo, CA, August 2001, pp. 149–160. ACM Press, New York (2001)