Elevator: Self-* and Persistent Hub Sampling Service in Unstructured Peer-to-Peer Networks

Mohamed Amine Legheraba, Maria Potop-Butucaru, Sébastien Tixeuil

Elevator: Self-* and Persistent Hub Sampling Service in Unstructured Peer-to-Peer Networks

Mohamed Amine LEGHERABA, Maria POTOP-BUTUCARU, and Sébatien TIXEUIL

Sorbonne Université

Abstract. We present Elevator, a novel algorithm for hub sampling in peer-to-peer networks, enabling the construction of overlays with a topology between a random graph and a star network, and networks that have both hubs and are resilient to failures. Our approach emerges from principles of preferential attachment, forming hubs spontaneously, offering an innovative solution for decentralized networks that can benefit use cases requiring a network with both low diameter and resilience to failures.

Keywords: Peer-to-peer networks · Peer sampling service · Hub sampling · Resilient networks · System design · Algorithms · Simulations.

1 Introduction

In recent years, the rise of decentralized systems such as blockchain [3] and federated learning [11] has spurred considerable interest in peer-to-peer (P2P) communication protocols. While existing P2P protocols have demonstrated significant utility across various applications, emerging demands for enhanced performance, scalability, and robustness necessitate the development of innovative solutions.

Peer-to-peer (P2P) protocols have undergone extensive research and development to facilitate efficient decentralized communication among networked devices. Foundational P2P protocols like Napster, Gnutella [6], and BitTorrent paved the way for distributed file sharing and content distribution across the Internet. Typically, P2P overlay networks are categorized as either structured (e.g. CAN [16], Chord [18], or Kademlia [10]) or unstructured (e.g. Gnutella [6]). More comprehensive details about peer-to-peer overlays can be found in recent surveys [9, 14].

Structured overlays come with a maintenance cost [9], and are more susceptible to Byzantine attacks (that is, attacks performed by the peers themselves) [14] and churn [9] (that is, the unexpected departure and arrival process of the peers). Unstructured networks exhibit advantages in resilience to node failures and adaptability to shifting network conditions [8], rendering them well-suited for dynamic and heterogeneous environments when compared to their structured counterparts. Their shortcomings are that the quality of services built on top of the network is difficult to assess.

Peer sampling. Peers within an unstructured overlay maintain a dynamic set of neighbors, often discovered through mechanisms like peer sampling [8], which enables nodes to gather and exchange information about other nodes in the network, and thus dictates the network topology. Existing peer sampling algorithms in the literature yield two types of topologies (random and power-law) that demonstrate favorable networking characteristics. Random graphs are built from gossip peer sampling algorithms, and are known to be resilient to churn [8]. Power-law (or scale-free) networks are built from algorithms that use the concept of preferential attachment and are known to have ultra-small diameter [4], which helps scalability. However, when considering the specific use case of federated learning, certain limitations emerge: *(i)* Gossip learning, based on gossip peer sampling, exhibits a slower convergence rate compared to centralized federated learning methodologies [7], and *(ii)* while power-law topologies theoretically offer improved convergence efficiency, prior research has predominantly focused on constructing networks adhering strictly to power-law distributions or implementing algorithms to restrict the proliferation of hubs [5] (that is, peers that are extremely well connected). Yet, for federated learning, the presence of hubs is advantageous, as these hubs facilitate rapid relay of machine learning models across the network, accelerating convergence rates. Nonetheless, conventional approaches relying on predefined hubs (e.g., super-peer-based topologies) are susceptible to attacks targeting static and well-defined hub nodes [12].

Hence, there exists a pressing need for a protocol that fosters the organic emergence of hubs within networks. The service outlined in this article is designed precisely for this purpose, allowing selected nodes to naturally ascend to hub status through a process we term "hub sampling". By enabling nodes to organically assume the role of hubs, our protocol aims to strike a balance between leveraging the efficiency of hub-based networks for applications like federated learning, while mitigating vulnerabilities associated with static hub designations.

Our contribution. Our primary goal is to develop a protocol that autonomously promotes nodes to act as hubs within unstructured peer-to-peer networks. To achieve this goal, we hybridize two fundamental concepts: *preferential* attachment, and *random attachment*. By integrating these two concepts, our protocol promotes a balanced network structure, where hubs emerge organically based on connectivity patterns, and yet adapt to dynamic network changes. This approach not only fosters robustness against failures and disruptions, but also maintains a low network diameter, facilitating efficient communication and information propagation. The parameter h, representing the desired number of hubs, allows for flexibility and control over the network's topology, enabling tailored configurations to suit specific application requirements and network environments. The rationale behind this initiative is rooted in the benefits of having hub nodes, particularly in applications such as federated learning, where efficient information dissemination is crucial. The existence of hubs facilitates faster network-wide communication compared to overlay networks structured in a random graph topology.

The structure of this article is organized as follows: Section 2 presents the hub sampling service altogether with its properties, its programming interface (API), and its implementation, the *Elevator algorithm*. Section 3 presents extensive simulations of Elevator, compared against three classical algorithms from the literature [8, 17, 20].

2 Hub sampling service

The key desired properties we expect from our protocol are *connectivity* (the overlay remains connected), *low-diameter* (for efficient communication), *convergence* (properties are obtained in an autonomous manner), *stability* (structural overlay properties are maintained throughout execution), and *robustness* (resilience to churn and targeted attacks). They will serve as metrics during simulation experiments to ascertain the efficacy of our algorithm.

2.1 Service API

The API of the hub sampling service mirrors that of classical peer sampling service [8], comprising two key methods: *(i) init()* that initializes the service on a given node, *i.e.*, initializes the list of outgoing connections of a node (Indeed, we assume that a given node starts connected to a random subset of nodes in the network, the actual initialization procedure being implementation-dependent), and *(ii) getPeer()* that returns a random peer address from the node list of peers.

The focus of this work is to present an implementation of the *getPeer()* method, Elevator, as a gossip-based algorithm, and to study the performance of its implementation. In addition to these two methods, we add a third method to the API called *getHub()* that returns a random hub. The *getHub()* method can be easily derived from *getPeer()* by filtering the output of *getPeer()* to only select the h nodes acting as hubs in the network. This method can be useful for applications that only need to contact a hub.

2.2 Preliminaries

In the context of our study, we consider an overlay network of interconnected nodes modeled as a directed graph. Communication within this network is bidirectional, corresponding to an underlying undirected graph that represents the physical network. Each node in this network possesses a unique address, akin to an IP address in the context of the Internet, serving as an abstract identifier of its identity. Nodes maintain a local list called *cache*, which contains addresses of other nodes, and represents their partial knowledge of the network's node set. The maximum size of this cache, denoted by parameter c, is uniform across all nodes. The cache is pivotal for peer sampling, as it serves as the basis for neighbor selection and information exchange. At the network's inception, nodes are initially connected to a random subset of nodes, forming what is known as a random k-out graph. Subsequently, new nodes joining the network also establish

connections with a random subset of existing nodes, a process that populates their cache and integrates them into the network. Given the decentralized nature of the network, peer sampling algorithms are designed to operate asynchronously, but we can refer to the idea of *cycles* of the protocol, as it is more convenient for the evaluation of protocols during simulations. During each cycle, every node initiates one execution of the peer sampling protocol, potentially updating its cache based on interactions with neighboring nodes. By leveraging cycles, we can analyze the convergence, performance, and robustness of peer sampling protocols under varying conditions and scenarios within the decentralized network environment.

2.3 Elevator core concepts

To achieve both robustness and a low network diameter, we integrate two fundamental concepts: preferential attachment and random attachment, each serving distinct yet complementary role in shaping the network topology.

Preferential Attachment. Drawing from the concept pioneered by Barabási and Albert [2], preferential attachment dictates that new connections in the network are established preferentially with nodes possessing a higher number of existing connections. In our adaptation, we modify this concept to elevate certain nodes to the status of hubs without requiring the network to continuously grow. Instead of new nodes joining and preferentially connecting to highly connected nodes, each existing node leverages information from its neighbors to identify and connect to the most frequently connected nodes (up to a predefined number h). This mechanism enables the organic emergence of hubs within the network, with selected nodes naturally assuming central roles based on their connectivity without any explicit distinction other than their number of incoming links.

Random Attachment. Inspired by gossip-based peer sampling algorithms [8, 17], random attachment ensures that nodes maintain connections with a representative and diverse subset of the network. This strategy promotes network robustness by preventing excessive clustering and dependency on specific nodes (hubs). When existing hubs disappear (e.g., due to failures or departure), other nodes within the network are opportunistically elevated to hub status, ensuring continuity and adaptability of the network topology over time.

Our target is to obtain a topology of the network that has the following properties: *(i)* There are h defined hubs, with h a parameter defined before the start of the network and common to all nodes, *(ii)* ignoring hubs, the distribution of the remaining connections is random, and *iii)* each node has c connections, consisting of h connections to hubs and c-h connections to random nodes.

Through simulation evaluation, we demonstrate in the sequel the effectiveness and advantages of our protocol with respect to state-of-the-art algorithms.

2.4 Elevator detailed description

The algorithm uses the following parameters and data structures:

- *Parameter* c: The maximum number of outgoing connections (its default value for all nodes is 20).

- *Parameter* h: The number of preferential attachment connections (its default value for all nodes is $c/2$).
- *Parameter* $maxsize\_buffer\_backward$: The maximum number of backward connections to send (its default value for all nodes is 100).
- *Structure cache*: The list of outgoing connections. The list is implemented as an array of size c. The list is initialized with random existing addresses (random connections to other nodes of the network).
- *Structure* $backward\_peers$: The list of other nodes that have tried to connect to the node. The list is implemented as a linked list (initially empty).

Additionally, we have three temporary structures: *(i)* $frequency\_map$ holds the frequency of occurrences for all neighbors of neighbors, implemented as a map ($node \rightarrow integer$), *(ii) preferred* holds the list of preferred nodes, implemented as a linked list, and *(iii)* $preferred\_backward$ holds the list of backward connections of the preferred nodes, implemented as a linked list.

The proposed protocol executes the following actions at each cycle: Each node retrieves the neighbor's list of their neighbors (*i.e.*, the neighbors at distance two). The node then builds an ordered list of the most frequent peers (the frequency map), and contacts the c most frequent nodes (called *preferred*). Each contacted node sends back to the contacting node a maximum of $maxsize\_buffer\_backward$ addresses from its backward list, maintained in the structure $backward\_peers$, and adds the contacting node to its backward list. The cache of the contacting node is then reset as an empty array. Then the node selects the h most frequent peers and c-h random peers from the list of backward peers of all preferred peers to fill its cache. If the cache is not full, the node adds random peers from the frequency map to the cache until the size of the cache is c (see Algorithm 1 and Algorithm 2 for detailed pseudocode of the algorithm).

3 Experimental evaluation

We evaluate our proposal by carrying out a simulation campaign. All simulations use the Java *PeerSim* simulator [13]. We have modified the simulator to add parallelism to accelerate computations. With Peersim, we implemented our algorithm Elevator, and state-of-the-art PROOFS [17] and Phenix [20] algorithms. Also, we used the implementation of Newscast provided by PeerSim. A detailed description of these algorithms can be found in Appendix A.

All simulations were run with a network of size n=1000. As the Phenix network needs a growing network to work, we started the Phenix algorithm with a network size of 20, and capped the size of the network to 1000. The simulations were run during 1000 cycles, and we repeated each simulation 100 times. All simulations were started with a network initialized as a k-out random graph, with k=c=20. All simulations were run on 16 vCPU, using 64G of memory, on a cluster composed of 10 servers of the following type:

Algorithm 1: Elevator Algorithm (active thread)

```
Data: initial peer list: cache
Data: cache size: c
Data: number of hubs desired: h
Data: initial backward list: backward_peers (empty)
Loop
    for peer ∈ backward_peers do
        if peer not responding then
            backward_peers.remove(peer)
    for peer ∈ cache do
        if peer not responding then
            cache.remove(peer)
    frequency_map ← {}
    for peer ∈ cache do
        peer_cache ← send(CACHE_REQUEST, peer)
        frequency_map ← frequency_map ∪ peer_cache
    preferred ← frequency_map.sortByFrequency().select(number = c)
    frequency_map.remove(preferred)
    preferred_backward ← {}
    for peer ∈ preferred do
        peer_backward_peers ← send(BACKWARD_REQUEST, peer)
        preferred_backward ←
          preferred_backward ∪ peer_backward_peers
    preferred.shuffle()
    preferred_backward.shuffle()
    cache ← {}
    cache ← preferred[0..h] + preferred_backward[0..c − h]
    while cache.size() < c do
        peer ← frequency_map.selectRandom()
        cache.append(peer)
```

Algorithm 2: Elevator Algorithm (background thread)

```
Data: max number of backward connections to send:
      maxsize_buffer_backward
Loop
    request, peer ← receive()
    if request = CACHE_REQUEST then
        send(cache, peer)
        backward_peers.add(peer)
    if request = BACKWARD_REQUEST then
        backward_peers.shuffle()
        send(backward_peers[: maxsize_buffer_backward], peer)
```

Machine	Memory	Processors	Cores
DELL PowerEdge XE8545	2 To	2 x AMD EPYC 7543	128 threads @ 2.80 GHz
DELL PowerEdge R750xa	2 To	2 x Intel Xeon Gold 6330	112 threads @ 2.00 GHz

We evaluated the following metrics: in-degree distribution, clustering coefficient, average shortest path length, and diameter. More details about those classical graph metrics can be found in Appendix B.

Figure 5 illustrates that the degree distributions of Newscast and PROOFS exhibit patterns akin to a normal distribution. We see similar results for Elevator, except for a distinct group of 10 hubs with an in-degree of 999. By contrast, the Phenix protocol's degree distribution conforms to a power-law distribution. PROOFS and Newscast maintain a low clustering coefficient during all simulations, as seen in Figure 1a. On the contrary, Phenix and Elevator have both a clustering coefficient of around 0.55. For Phenix, the value is related to the power-law distribution of in-degree, and for Elevator, the value is linked to the presence of hubs, that are connected to everyone, and this automatically increases the value of the coefficient. As we can see in Figure 1b, Elevator has a very low average path length, with a value below 2. This value is due to the presence of hubs in the network, that permit to have a maximum distance of 2 between 2 nodes. Phenix is a bit better, with a value slightly above 1.9. PROOFS is very close, with a value around 2.15 and Newscast is a bit below 2.6. All these values are very good and thus we need to compute the diameter to discriminate between algorithms. In Figure 1c, we see that Elevator gives a network with a diameter almost equal to the average path length, with a value almost equal to 2. Again, this value is due to the presence of hubs in the network. The Phenix algorithm yields similar results. This is better than PROOFS and Newscast, which output respectively 3 and 4 for this metric.

We also compared the algorithms according to their resilience to crashes, churn, and attacks on hubs, as shown below. Additional results and the accompanying figures are included in the Appendix C.

3.1 Resilience to crashes

We analyze the performance of the four algorithms when the network suffers crashes.

Resilience to sparse crashes. In Figure 7, we consider the biggest weakly connected cluster of the network after the run of each algorithm, and we then remove one by one all nodes inside it, and compute the number of nodes outside this cluster. As we can see in Figure 7 for all algorithms, we don't observe outsider nodes until we remove 80% of the nodes.

Resilience to massive crashes. To simulate a brutal failure we disconnecte 50% of the nodes in the middle of the simulation, *i.e.*, in this case, we have disconnect 500 nodes at cycle 500 (as there are 1000 nodes in total, and 1000 cycles). As we

can see in Figures 5b and 6b, the performance of Elevator is not affected, as the in-degree distribution is still the same, and we have 10 hubs with an in-degree of 499. The degree distribution is also the same for Newscast and PROOFS. For Phenix, the degree distribution remains the same, with values going to a max of 999, even if there are only 500 nodes in the network. It's because the nodes have kept in their cache the addresses of (old) nodes who are no longer in the network. In Figure 2a, the clustering coefficient evolution shows that it is not affected by the crashes, as we have almost the same results as those obtained without a crash. The same observation holds for the average path length and the diameter, as we can see in Figures 3a and 4a.

3.2 Resilience to churn

We now analyze the performance of the four algorithms when the network is subject to churn. To simulate churn, we disconnecte 10% of the nodes at each cycle, and replace them with the same amount of new nodes, each connected to 20 nodes uniformly at random. The churn occurs during 500 cycles, between cycle n°250 and cycle n°750. As the Phenix algorithm needs a growing network to work, the way we implement churn differs. Following previous work [20], in the case of Phenix, we implement churn having the number of removed nodes less than the number of added nodes at each cycle, assuming nodes are removed following a normal distribution $\mathcal{N}(0, 1)$, for all cycles of the simulation.

As we can see in Figures 5c and 6c, the in-degree distribution of Elevator remains the same, with 10 hubs. PROOFS seems affected by churn, as the mean degree distribution goes to 10 instead of 20 without churn. In Figure 2b we can observe that we have almost the same results as the results obtained without churn for the clustering coefficient, except Phenix which seems affected by churn, as its clustering coefficient varies greatly, which is probably because the coefficient decreases a lot if the nodes affected by churn are the ones with a high in-degree. For the average path length, PROOFS is the most affected, with a value going from 2.25 without churn to a value of 2.5 with churn, as we can see in Figure 3b. In Figure 4b, we can see that the diameter varies with churn, with a mean going up to 2.5 instead of 2.0, but the values for Phenix and Elevator remain below the ones of Newscast and PROOFS.

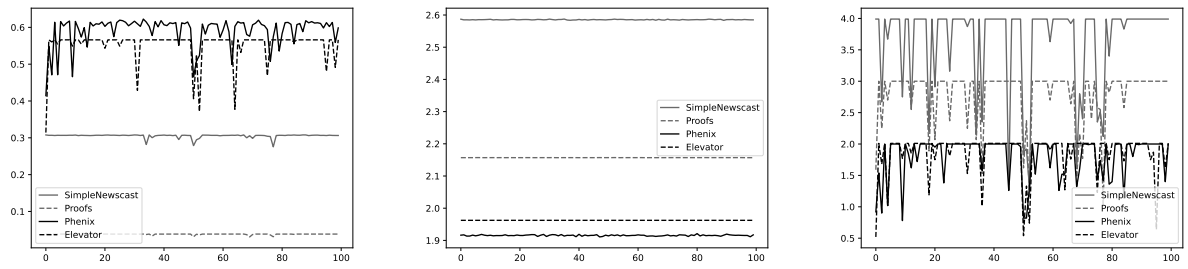
3.3 Resilience to hub-targeted attacks

We hereby analyze the performance of the four algorithms after a targeted attack on the hubs during the execution of the simulation. To simulate a hub-targeted attack, we disconnected 10 nodes that have the highest in-degree in the middle of the simulated scenario.

Logically, Newscast and PROOFS are not affected by the attack, as there are no hubs in the networks built by these algorithms. For Elevator, as we can see in Figure 5d and 6d, the in-degree distribution remains similar, with 10 high-in-degree peers that have each an in-degree of 989. We are thus confident in the capacity of our algorithm to promote new nodes to the position of hubs if the

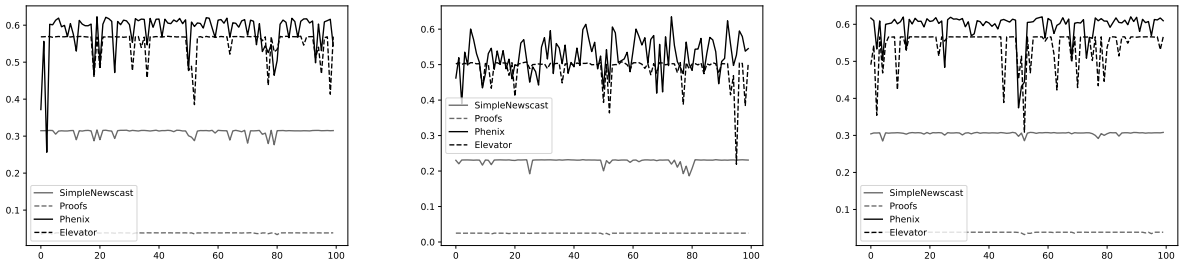
previous hubs were disconnected. In Figure 2c we can see that we have almost the same results as the results obtained without crashes for the clustering coefficient, except for the clustering coefficient dropping from 0.55 to 0.3 in the middle of the simulation for Elevator, which is logical as the 10 hubs are disconnected. The drop is only temporary, as the value goes back to 0.55 almost immediately. For the average path length and the diameter there is no impact, as we can see in Figure 3c and 4c.

Summary. In Figure 8 we compare the in-degree distribution of the network after the run of the Elevator algorithm for a various number of hubs 8a, and also for each context of simulation 8b. The shape of the degree distribution remains consistent across different hub counts, except for a scenario with 20 hubs where nodes exclusively connect to these hubs (resulting in a multi-star topology). This phenomenon aligns with the prescribed number of preferred connections ($h = c = 20$), where nodes exclusively link to elevated hub nodes, omitting random connections entirely. The shape of distribution also remains consistent across failure contexts. In Figure 9, we compare Elevator across all contexts for the different metrics, and we can see that there are not many variations in values, as expected from the definition of our protocol and as seen in previous comparative analyses presented above.



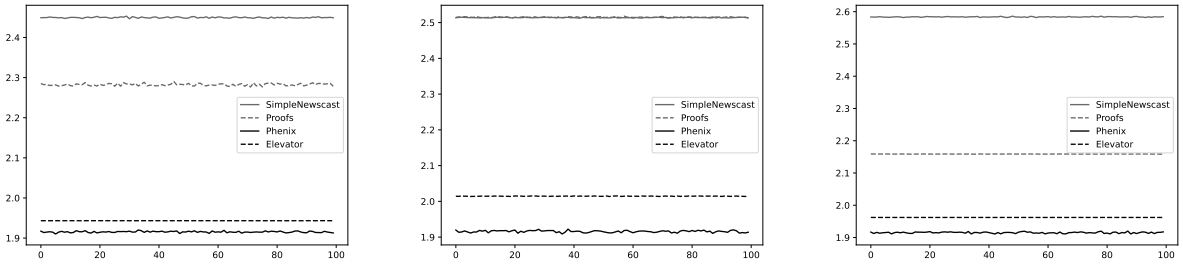
(a) Clustering coefficient (b) Average path length (c) Diameter

Fig. 1: Metrics computed during the simulation (no failures), for each algorithm, every 10 cycles.



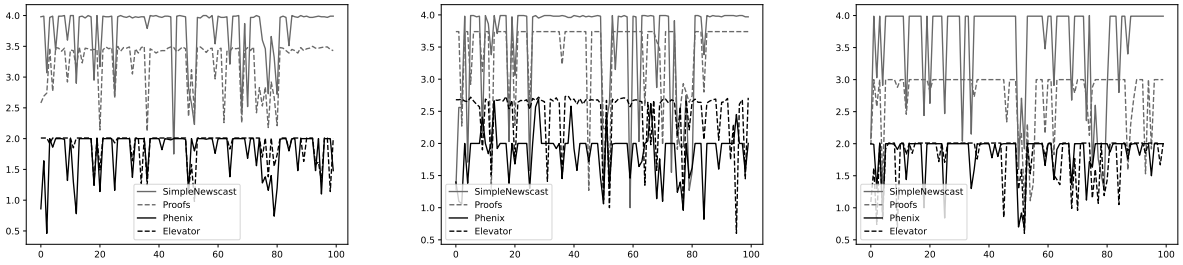
(a) With a 50% crash (b) With churn (c) With a crash of all hubs

Fig. 2: Clustering coefficient for each algorithm, every 10 cycles.



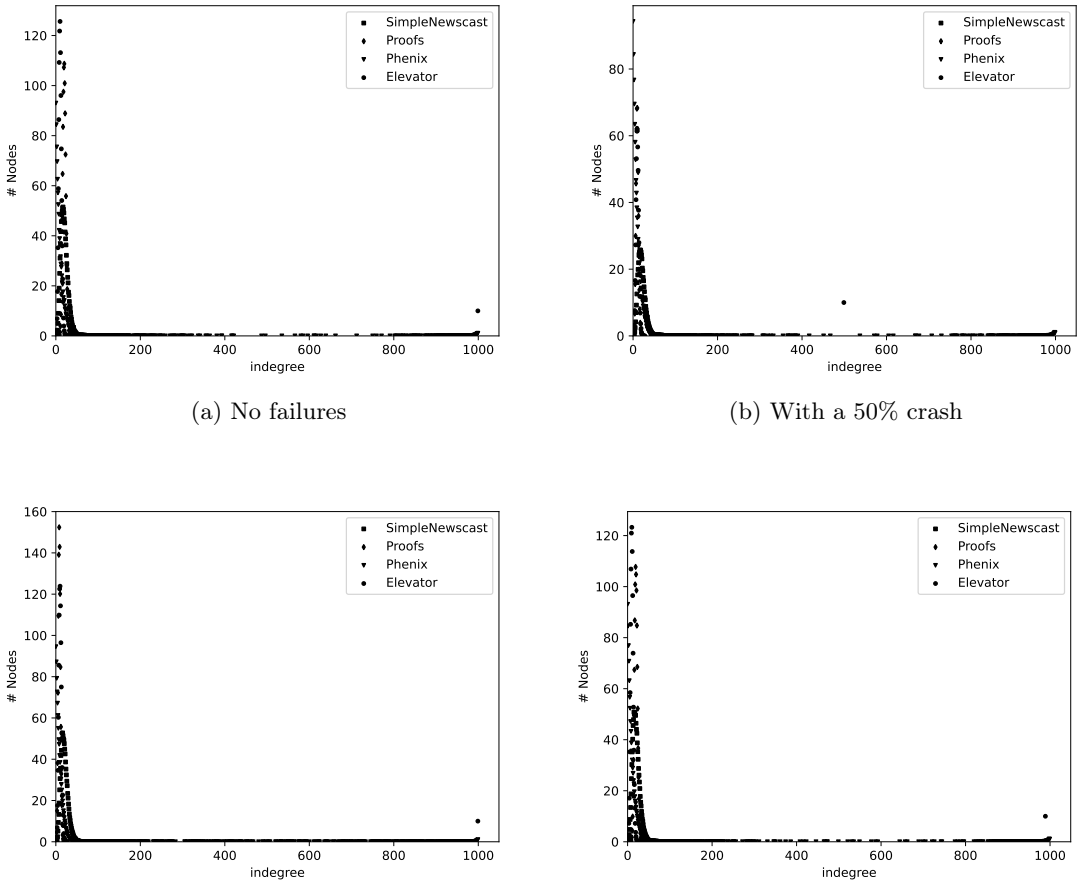
(a) With a 50% crash (b) With churn (c) With a crash of all hubs

Fig. 3: Average path length for each algorithm, every 10 cycles.



(a) With a 50% crash (b) With churn (c) With a crash of all hubs

Fig. 4: Diameter of the graph, for each algorithm, every 10 cycles.



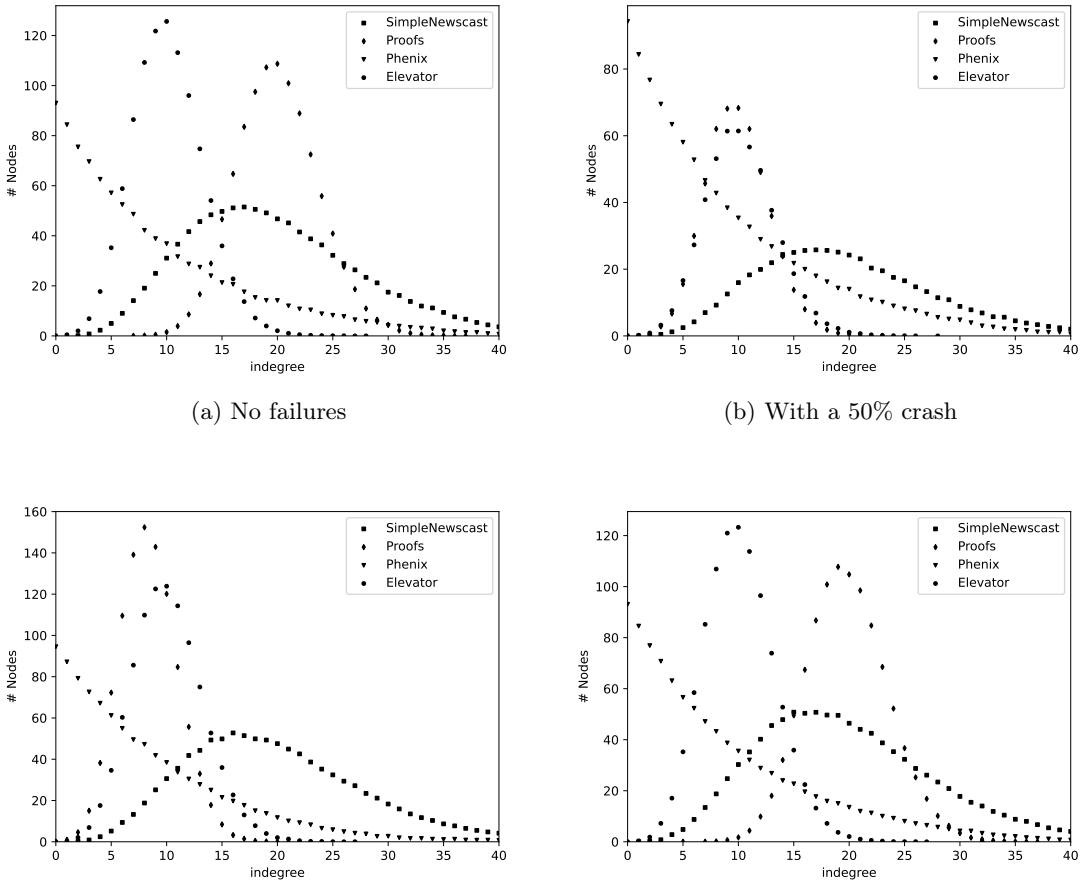
(a) No failures

(b) With a 50% crash

(c) With churn

(d) With a crash of all hubs

Fig. 5: In-degree distribution of the network, after the run of the algorithm for 1000 cycles, for each algorithm.



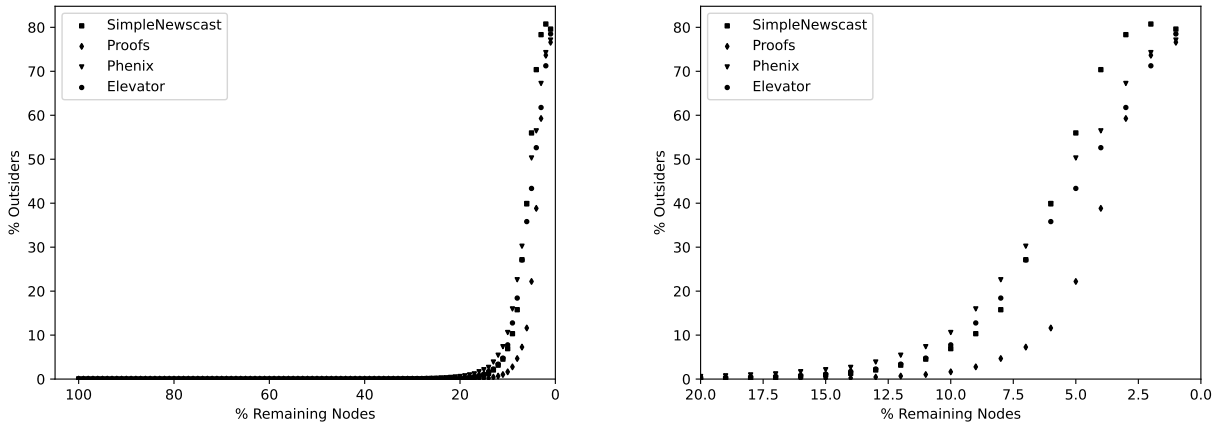
(a) No failures

(b) With a 50% crash

(c) With churn

(d) With a crash of all hubs

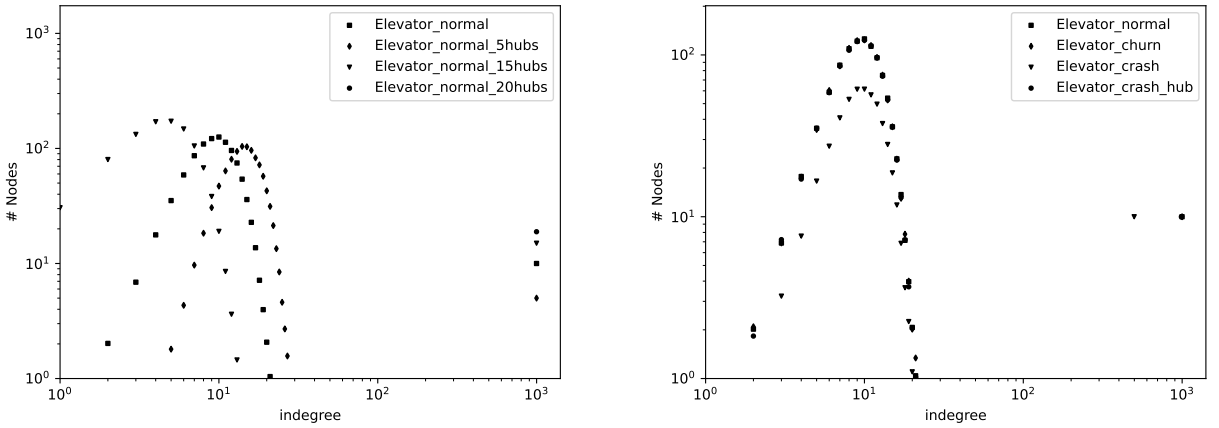
Fig. 6: In-degree distribution of the network, after the run of the algorithm for 1000 cycles, for each algorithm, zoom on the beginning of distribution.



(a) Removing of all the nodes

(b) Zoom on the last 20%

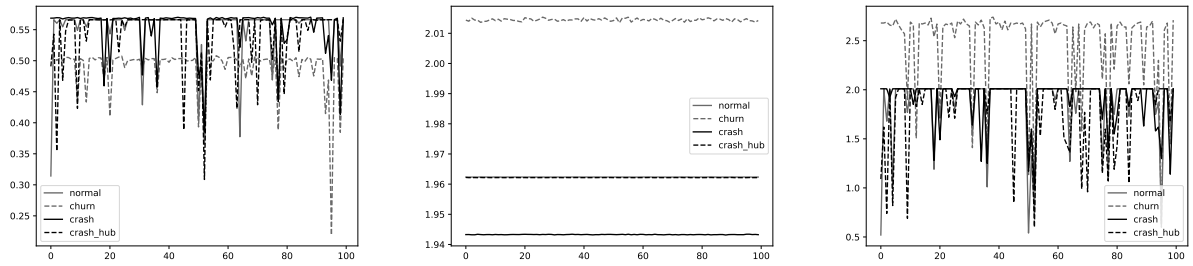
Fig. 7: Analysis of the removal of all the nodes of the network one by one, and observing the number of nodes outside the biggest weakly connected cluster.



(a) Variable number of hubs (5 hubs, 10 hubs, 15 hubs, 20 hubs), no failures

(b) Each context (no failures, 50% crash, churn and hub-targeted attack).

Fig. 8: In-degree distribution of the network, after the run of the Elevator algorithm for 1000 cycles.



(a) Clustering coefficient (b) Average path length (c) Diameter

Fig. 9: Metrics computed during the simulation, after the run of the Elevator algorithm, for all contexts (normal, churn, 50% crash, hub-targeted attack), every 10 cycles.

4 Conclusion

We proposed a novel peer sampling algorithm, Elevator, designed for unstructured P2P networks, which facilitates the organic promotion of specific nodes to serve as hubs. Our simulations confirm that the Elevator algorithm successfully maintains network connectivity, constructs networks with low diameters, achieves stability with a defined number of hubs (denoted as h), and demonstrates resilience against crashes, churn, and targeted attacks on hubs. We anticipate that this work will pave the way for a new category of algorithms known as "hub sampling algorithms", which could hold significant relevance for specific decentralized applications. For instance, such algorithms may accelerate the transmission of machine learning models in federated learning scenarios. While our current study does not delve into these specific use cases, we envision exploring federated learning applications within this network paradigm in future investigations.

References

[1] Alexandros Antonov and Spyros Voulgaris. "SecureCyclon: Dependable Peer Sampling". In: *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2023, pp. 1–12.

[2] Albert-Laszlo Barabâsi et al. "Evolution of the social network of scientific collaborations". In: *Physica A: Statistical mechanics and its applications* 311.3-4 (2002), pp. 590–614.

[3] Nakamoto S Bitcoin. *Bitcoin: A peer-to-peer electronic cash system*. 2008.

[4] Reuven Cohen and Shlomo Havlin. "Scale-free networks are ultrasmall". In: *Physical review letters* 90.5 (2003), p. 058701.

[5] Suyong Eum, Shin'ichi Arakawa, and Masayuki Murata. "Self-organizing scale free topology for peer-to-peer networks". In: *2009 IEEE Globecom Workshops*. IEEE. 2009, pp. 1–6.

[6] Justin Frankel. "The Gnutella protocol specification v0. 4". In: *http://www. clip2. com/gnutellaprotocol04. pdf* (2003).

[7] István Hegedűs, Gábor Danner, and Márk Jelasity. "Decentralized learning works: An empirical comparison of gossip learning and federated learning". In: *Journal of Parallel and Distributed Computing* 148 (2021), pp. 109–124.

[8] Márk Jelasity et al. "Gossip-based peer sampling". In: *ACM Transactions on Computer Systems (TOCS)* 25.3 (2007), 8–es.

[9] Apostolos Malatras. "State-of-the-art survey on P2P overlay networks in pervasive computing environments". In: *Journal of Network and Computer Applications* 55 (2015), pp. 1–23.

[10] Petar Maymounkov and David Mazieres. "Kademlia: A peer-to-peer information system based on the xor metric". In: *International Workshop on Peer-to-Peer Systems*. Springer. 2002, pp. 53–65.

[11] Brendan McMahan et al. "Communication-efficient learning of deep networks from decentralized data". In: *Artificial intelligence and statistics*. PMLR. 2017, pp. 1273–1282.

[12] Alberto Montresor. "A robust protocol for building superpeer overlay topologies". In: *Proceedings. Fourth International Conference on Peer-to-Peer Computing, 2004. Proceedings.* IEEE. 2004, pp. 202–209.

[13] Alberto Montresor and Márk Jelasity. "PeerSim: A Scalable P2P Simulator". In: *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*. Seattle, WA, Sept. 2009, pp. 99–100.

[14] Ashika R Naik and Bettahally N Keshavamurthy. "Next level peer-to-peer overlay networks under high churns: a survey". In: *Peer-to-Peer Networking and Applications* 13.3 (2020), pp. 905–931.

[15] Róbert Ormándi, István Hegedűs, and Márk Jelasity. "Gossip learning with linear models on fully distributed data". In: *Concurrency and Computation: Practice and Experience* 25.4 (2013), pp. 556–571.

[16] Sylvia Ratnasamy et al. "A scalable content-addressable network". In: *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. 2001, pp. 161–172.

[17] Angelos Stavrou, Dan Rubenstein, and Sambit Sahu. "A lightweight, robust P2P system to handle flash crowds". In: *IEEE Journal on Selected Areas in Communications* 22.1 (2004), pp. 6–17.

[18] Ion Stoica et al. "Chord: a scalable peer-to-peer lookup protocol for internet applications". In: *IEEE/ACM Transactions on networking* 11.1 (2003), pp. 17–32.

[19] Spyros Voulgaris, Daniela Gavidia, and Maarten Van Steen. "Cyclon: Inexpensive membership management for unstructured p2p overlays". In: *Journal of Network and systems Management* 13 (2005), pp. 197–217.

[20] Rita H Wouhaybi and Andrew T Campbell. "Phenix: Supporting resilient low-diameter peer-to-peer topologies". In: *IEEE INFOCOM 2004*. Vol. 1. IEEE. 2004.

A Description of PROOFS, Newscast and Phenix algorithms

As our goal is to present our new hub sampling algorithm and compare it to previous peer sampling algorithms, we will (briefly) present three peer sampling algorithms (PROOFS, Newscast, and Phenix). We chose to compare our proposed algorithm to these three algorithms as they are widely used in the literature. Newscast is used for gossip learning[15], PROOFS is a foundational algorithm, as Secure Cyclon[1], one of the latest peer sampling algorithm in the literature, is based on Cyclon[19], itself based on PROOFS. Phenix is interesting as it has especially been conceived to be resilient to failures and Byzantine attacks and also to construct networks that have a low diameter.

The PROOFS algorithm: The PROOFS algorithm, as presented in [17] is a very simple algorithm used to create a peer sampling service. At each cycle, each node initiates a neighbor exchange (or shuffling) with another peer q chosen at random. The peer selects a random subset of size l (the shuffle length, a global parameter) and sends this subset to q. Upon reception of the subset, the node q also selects a random subset and sends it to p. When the node receives the subset of q, it replaces the previous entry in its cache, starting with the empty cache slots (if any) and then replacing entries previously sent to q. The parameters of the algorithm are c, the size of the list of outgoing connections, and l, the shuffle length, i.e. the number of outgoing connections exchanged with a peer during a neighbor exchange. The cache list is implemented as an array of size c. The list is initialized with random values (random connections to other nodes of the network).

Algorithm 3: PROOFS algorithm (active thread)

Data: initial peer list: *cache*
Data: cache size: c
Data: shuffle length: l
Data: node address: p
Loop
  $subset \leftarrow selectRandomSubset(cache, l)$
  $q \leftarrow selectRandom(subset)$
  $subset.remove(q)$
  $subset.add(p)$
  $send(q, subset)$
  $subset_q \leftarrow receive(q)$
  $subset_q.remove(p)$
  $subset_q.removeAll(cache)$
  $cache \leftarrow subset_q$

Algorithm 4: PROOFS algorithm (background thread)

Data: peer list: cache
Data: shuffle length: l
Data: node address: p
Loop
  $q, subset_q \leftarrow receive()$
  $subset \leftarrow selectRandomSubset(cache, l)$
  $send(q, subset)$
  $subset_q.remove(p)$
  $subset_q.removeAll(cache)$
  $cache \leftarrow subset_q$

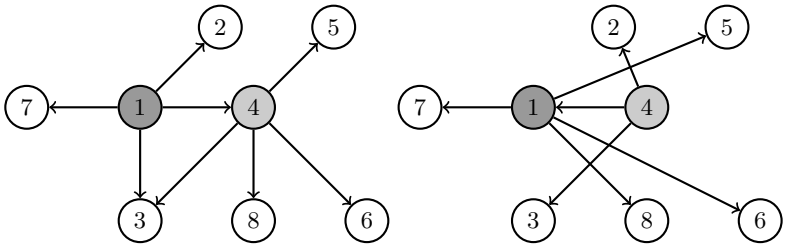


Fig. 10: Before and after the execution of the PROOFS algorithm. The node 1 sends its address alongside the address of 2 and 3 to the node 4. Node 4 sends back the addresses of nodes 5,6 and 8. Node 1 replaces the connection to 2, 3, and 4 with connections to 5,6 and 8. Node 4 drops connection to 5,6, and 8 and connect to nodes 2 and 1 (it is already connected to 3).

The goal of the algorithm is to produce a network that is "well-mixed", in the sense that after enough shuffling operations, the node's neighbors are essentially drawn at random from the set of all peers.

The Newscast algorithm: The Newscast algorithm[8] is similar to PROOFS but is more generic and adds the idea of "age" for the node descriptors. The age of the node descriptors is incremented at each cycle. The goal is to create a peer sampling service that allows each node of the network to connect to a random subset of the nodes in the network. The parameters of the algorithm are c, the size of the list of outgoing connections, the mode of the peer selection (random or tail, but for simulations we only used random), and the mode of view propagation. The mode of view propagation can be push, pull, or push-pull, as described below:

– Push strategy: At each cycle, a node will send its knowledge to the selected node

– Pull strategy: At each cycle, a node will ask for knowledge from the selected node and wait for the answer.

– Push-Pull strategy: At each cycle, a node will both use a Push and a Pull strategy.

As the push-pull mode is the most efficient[8], this is the mode we will present and the one that we used in our simulations. The cache list is implemented as an array of 2-tuple of size c. The two elements of the tuple are the node descriptor and the associated age of the descriptor. The list is initialized with random values for the node descriptors (random connections to other nodes of the network) and with 0s for the age of the node descriptors. At each cycle, each node initiates an exchange of membership information with a neighbor chosen at random. The node sends a buffer that contains $\frac{c}{2} - 1$ random node descriptors from its cache to the other node, with c the parameter representing the size of the cache. The other node replies to the message with a similar message also containing a buffer with $\frac{c}{2} - 1$ nodes descriptors. The node then merges the received buffer with its cache and filters the elements (removing the duplicates, the older elements, and finally removing the sent elements) to achieve a cache of the same size as before. If there are still too many elements, the algorithm removes elements of the cache at random until the length of the *cache* is c.

Algorithm 5: Newscast algorithm (active thread)

Data: initial peer list: cache
Data: cache size: c
Data: node address: p
Loop
  $q \leftarrow selectRandom(cache)$
  $buffer =\leftarrow \{\}$
  $buffer.append(address = p, age = 0)$
  $cache.permute()$
  $buffer.append(view.head(c/2 - 1))$
  $send(q, buffer)$
  $buffer_q \leftarrow receive(q)$
  $cache.append(buffer_q)$
  $cache.removeDuplicates()$
  $cache.removeOldItems()$
  $cache.removeHead()$
  $cache.removeRandom()$
  $cache.IncrementAllItemsAge()$

Algorithm 6: Newscast algorithm (background thread)

```
Data: peer list: cache
Data: node address: p
Loop
    q, buffer_q ← receive()
    buffer = newbuffer()
    buffer.append(address = p, age = 0)
    cache.permute()
    buffer.append(view.head(c/2 − 1))
    send(q, buffer)
    cache.append(buffer_q)
    cache.removeDuplicates()
    cache.removeOldItems()
    cache.removeHead()
    cache.removeRandom()
    cache.IncrementAllItemsAge()
```

As with PROOFS, the Newscast algorithm allows the creation of a network that has the same behavior as a random graph. In particular, this allows the network to be very resilient to failures and churn, as explained in [8].

The Phenix algorithm: The Phenix algorithm[20] is a peer sampling algorithm that differs from PROOFS and Newscast in the sense that the goal of the authors is to create an algorithm that is both resilient to failures and with a low-diameter. The algorithm is inspired by the concept of preferential attachment[2] and constructs a network with a topology that is close to a power-law. Contrary to the two previous algorithms, the Phenix algorithm only executes once for each node, when the node enters the network. The node splits its cache in 2 parts G_{random} and $G_{friends}$. Then it connects directly to the nodes in G_{random} and asks for the list of neighbors for each node in G_{friend} and adds them to the list $G_{candidates}$. Each node in G_{friend} also sends a ping message to all its neighbors and all neighbors add the new node to their Γ list, to prevent crawling from malicious nodes. The new node then sorts this list of distance two neighbors ($G_{candidates}$) and selects the s more frequent nodes and connects to them ($G_{preferred}$). When a node receives a connection request from a new node, it will increment its internal counter and create a backward connection with this node if its counter's value is greater than the γ constant. The parameters of the algorithm are c, the number of outgoing connections, τ, the number of cycles a node is kept in the gamma list (fixed at 10), γ, the constant limiting the number of backward connections (fixed to 20, thus a backward connection is created for 20 in-going connections) and s, the number of preferential connections, chosen to the value of $c/2$ for our simulations. The cache list is implemented as an array of size c. The list is initialized with random values (random connections to other nodes of the network). The Γ list is implemented as a linked list and initialized as an empty list.

Algorithm 7: Phenix algorithm (active thread)

Data: cache size: c
Data: initial peer list: cache
Data: initial backward list: $backward\_peers$ (empty)
Data: number of preferential connections: s

$G_{random}, G_{friend} \leftarrow split(cache)$
$cache \leftarrow \{\}$
$cache.append(G_{random})$
$G_{candidates} \leftarrow \{\}$
for $peer \in G_{friend}$ **do**
    $neighbor\_list \leftarrow send(peer, CACHE\_REQUEST)$
    $G_{candidates} \leftarrow G_{candidates} \cup neighbor\_list$
$sort(G_{candidates})$
$G_{preferred} \leftarrow G_{candidates}[0..(s-1)]$
for $peer \in G_{prefered}$ **do**
    $send(peer, CONNEXION\_REQUEST)$
$cache.append(G_{preferred})$

Algorithm 8: Phenix algorithm (background thread)

Data: peer list: cache
Data: gamma list: Γ
Data: initial backward list: $backward\_peers$ (empty)
Data: number of preferential connections: s (fixed at c/2)
Data: internal counter for backward connections: c_m (start at 0)
Data: backward connections constant: γ (fixed at 20)

Loop
    $\Gamma.removeOldItems()$
    $request, peer \leftarrow receive()$
    if $request = CACHE\_REQUEST$ **then**
        $send(cache, peer)$
        for *node in cache* **do**
            $send(node, PING\_REQUEST)$
    if $request = PING\_REQUEST$ **then**
        $\Gamma.add(peer)$
    if $request = CONNEXION\_REQUEST$ **then**
        c_m++
        if $c_m >= \gamma$ **then**
            $backward\_peers.add(peer)$
            $c_m \leftarrow c_m - \gamma$

To allow for the idea of preferential attachment to work, the network needs to be initialized with a small number of nodes (the number is set to 20 in [20] and we have chosen the same value for our simulations) and nodes are added

progressively, with the number of nodes added at each cycle is drawn from a normal distribution $\mathcal{N}(2,1)$. The constructed network is a scale-free network, with a topology following a power law.

B Metrics used: Degree Distribution, Clustering, Average Path Length and Diameter

B.1 Degree distribution

The *indegree (resp outdegree) distribution* of a network represents the probability distribution of these indegrees (resp outdegrees) over the whole network.

- A network that follows a random graph distribution (Erdős–Rényi model) should have a degree distribution that follows the probability $P(k) = \binom{n-1}{k} p^k (1-p)^{n-1-k}$
- A network that follows a power law (Barabási-Albert model) should have a degree distribution that follows the following probability $P(k) = Ck^{-\gamma}$

Indeed, observing the degree distribution should tell us if our algorithm creates a network with a topology closer to a random graph or one closer to a power-law.

B.2 Clustering coefficient

A random graph tends to have a low clustering coefficient, and a network with a lot of hubs will have a higher clustering coefficient. The *clustering coefficient* of a node is the number of edges between the neighbors of the node divided by the number of all possible edges between those neighbors. Intuitively, we can think of this coefficient as the measure of the degree to which nodes in a graph tend to cluster together (neighbors of the node are also neighbors of each other).

$$C_i = \frac{2e_i}{k_i(k_i - 1)}$$

Where:

- e_i is the number of closed triangles containing node i.
- k_i is the degree of node i, which is the number of links (edges) connected to that node.

$$C = \frac{1}{n} \sum_{i=1}^{n} C_i$$

B.3 Average Path Length

As our goal is to have an algorithm that constructs a network that disseminates information in the network, our algorithm must produce a network topology with a low average path length. The average path length of each node was computed using the Floyd–Warshall algorithm.

The average path length is the average of the shortest path lengths over all pairs of nodes in the graph.

$$a = \sum_{\substack{s,t \in V \\ s \neq t}} \frac{d(s,t)}{n(n-1)}$$

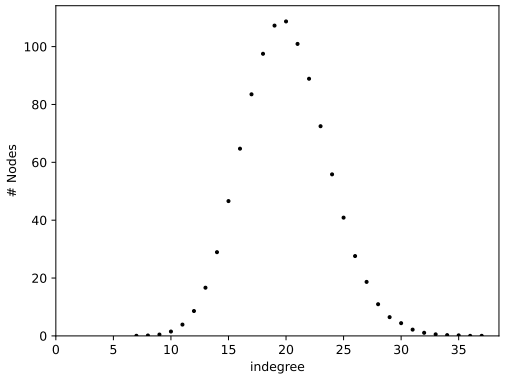
B.4 Diameter

The diameter of a graph is a measure of the longest distance between any two vertices (nodes) in the graph, measured in terms of the number of edges. In other words, the diameter of a graph is the maximum shortest path between any pair of nodes in the network.

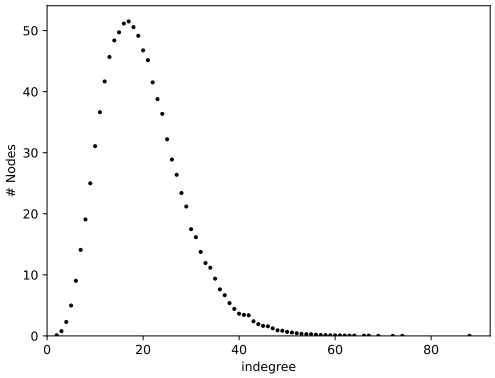
$$\operatorname{diam}(G) = \max_{u,v \in V} \mathrm{d}(u,v)$$

While the average path length provides a basic measure of information dissemination efficiency in algorithms, it may overlook disparities in dissemination speed across different nodes within the network. An algorithm could potentially have a favorable average path length but still exhibit uneven dissemination speeds among nodes due to varying distances. Calculating the network's diameter, however, offers a more comprehensive assessment.

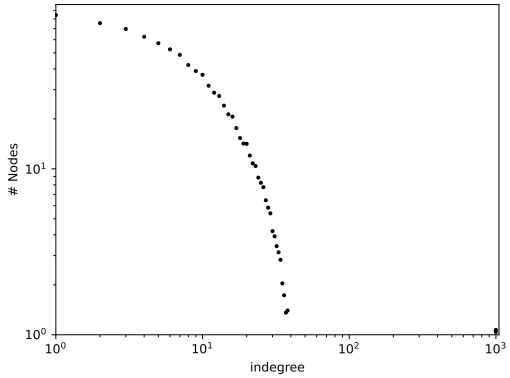
C Additional results from simulations



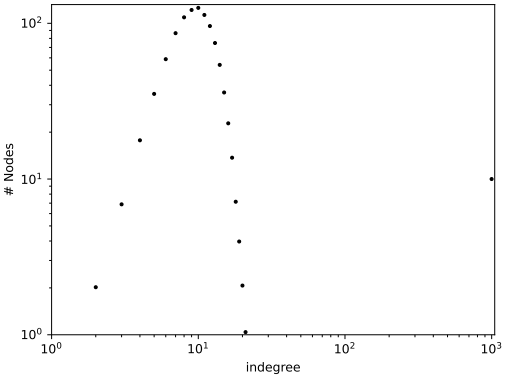
(a) PROOFS



(b) Newscast

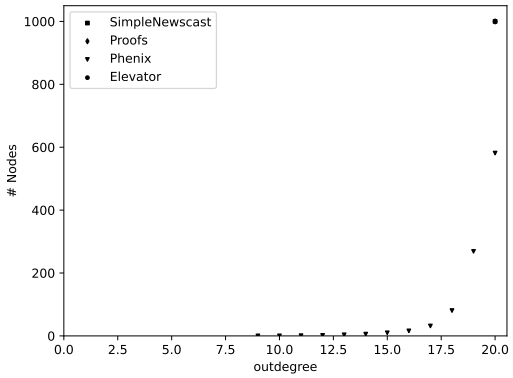


(c) Phenix (loglog scale)

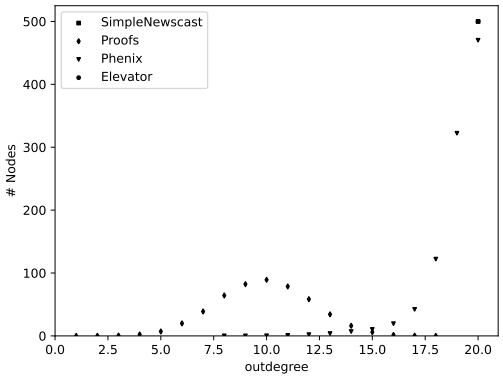


(d) Elevator (loglog scale)

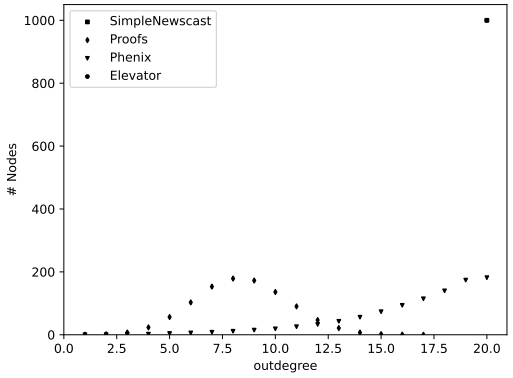
Fig. 11: In-degree distribution of the network, after the run of the algorithm for 1000 cycles.



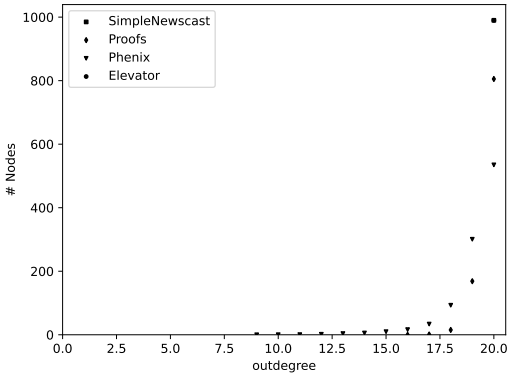
(a) Without failures



(b) With a 50% crash

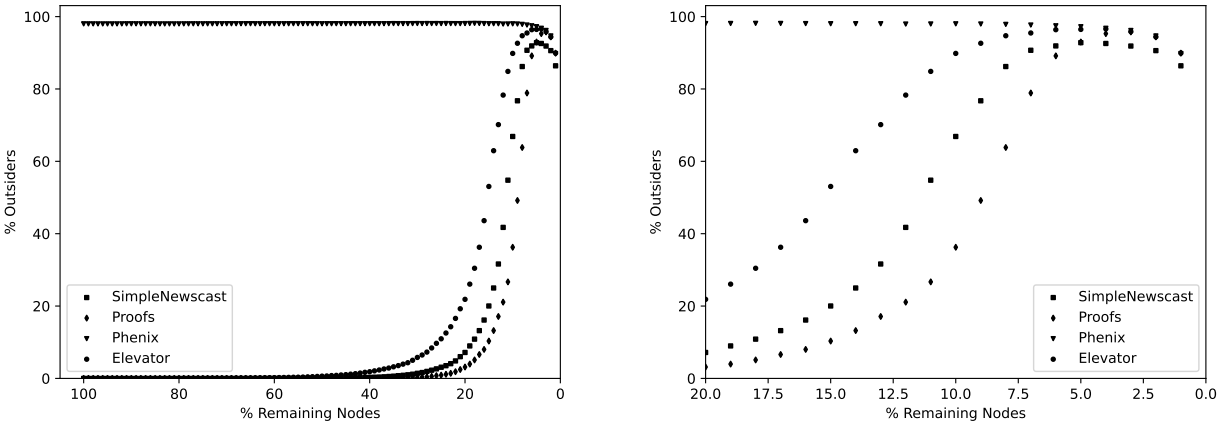


(c) With churn



(d) With a hub-targeted attack

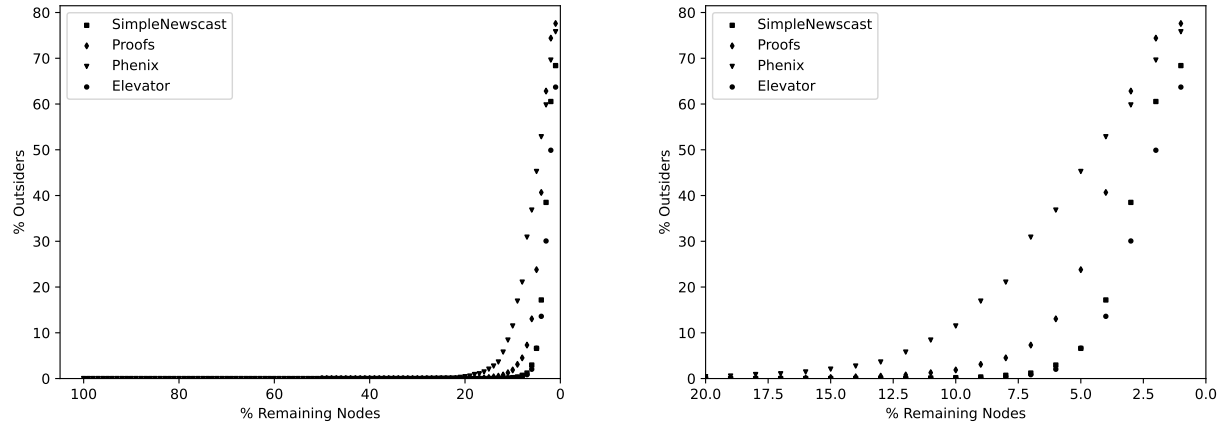
Fig. 12: Out-degree distribution of the network, after the run of the algorithm for 1000 cycles, for each algorithm.



(a) Removing of all the nodes

(b) Zoom on the last 20%

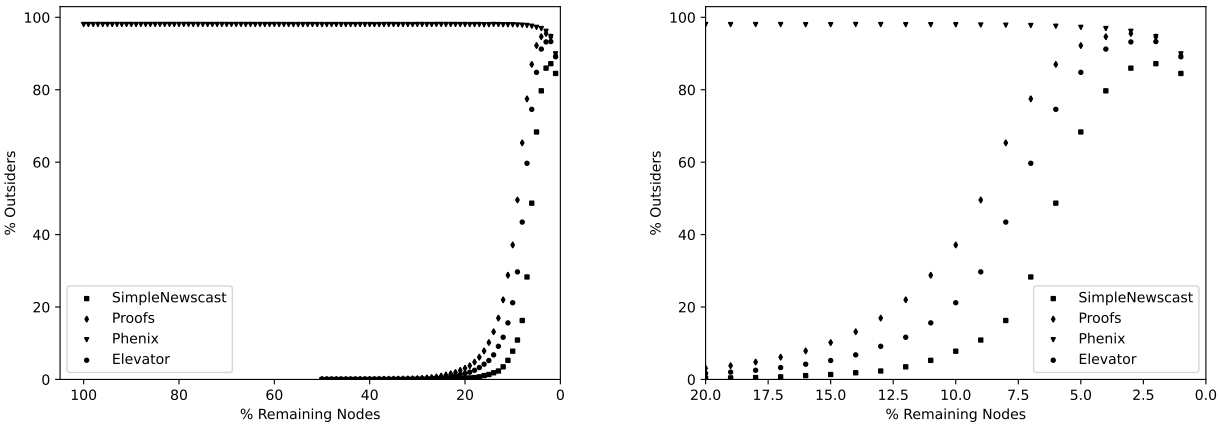
Fig. 13: Analysis of the removal of all the nodes of the network one by one, and observing the number of nodes outside the biggest strongly connected cluster.



(a) Removing of all the nodes

(b) Zoom on the last 20%

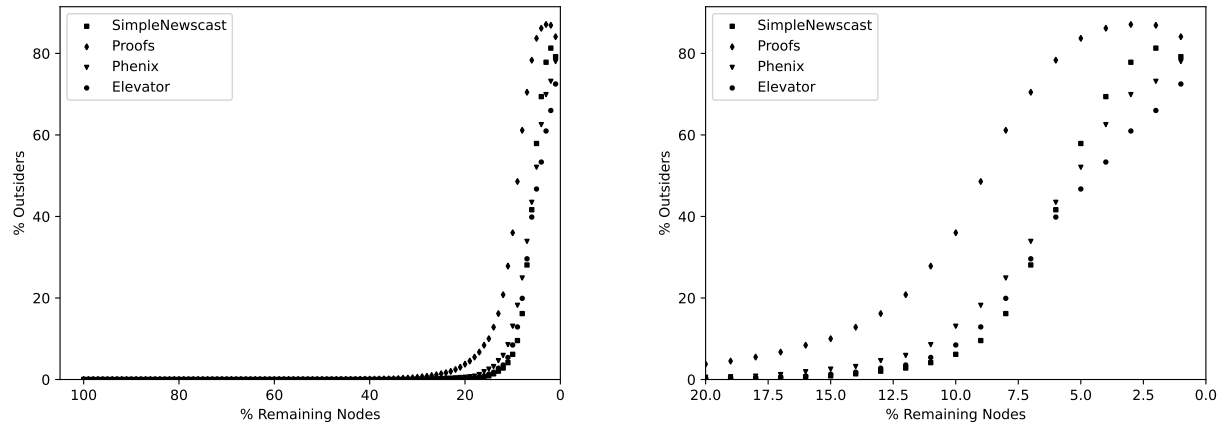
Fig. 14: Analysis of the removal of all the nodes of the network one by one, and observing the number of nodes outside the biggest weakly connected cluster, with a 50% crash.



(a) Removing of all the nodes

(b) Zoom on the last 20%

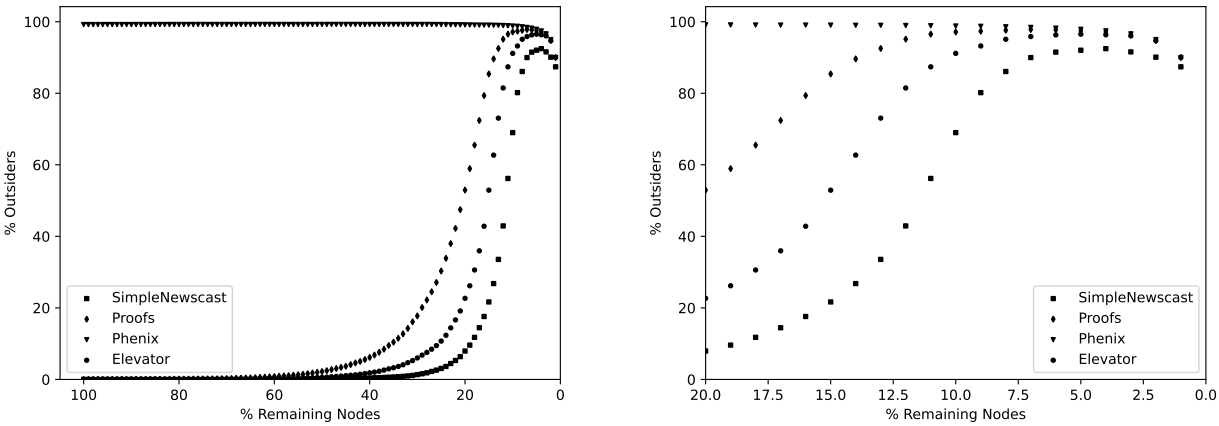
Fig. 15: Analysis of the removal of all the nodes of the network one by one, and observing the number of nodes outside the biggest strongly connected cluster, with a 50% crash.



(a) Removing of all the nodes

(b) Zoom on the last 20%

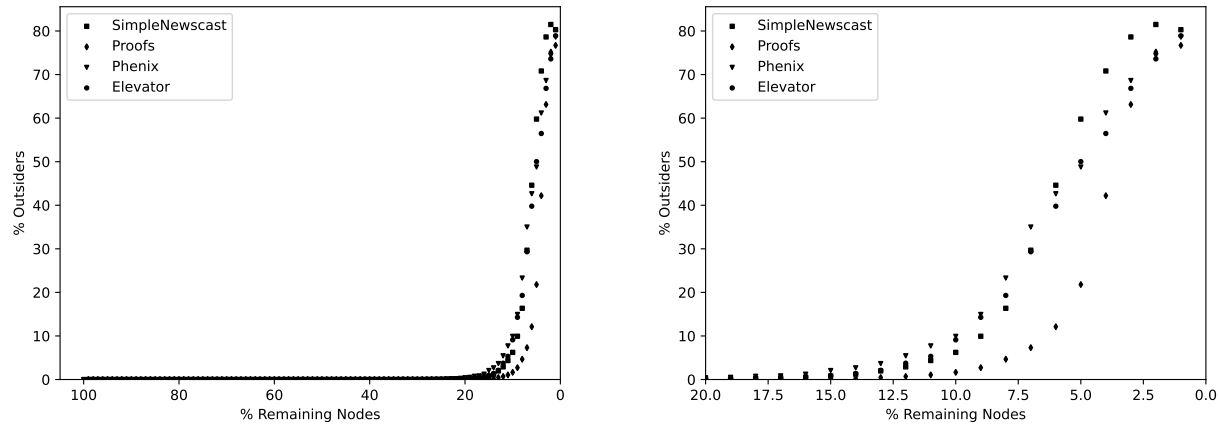
Fig. 16: Analysis of the removal of all the nodes of the network one by one, and observing the number of nodes outside the biggest weakly connected cluster, with churn.



(a) Removing of all the nodes

(b) Zoom on the last 20%

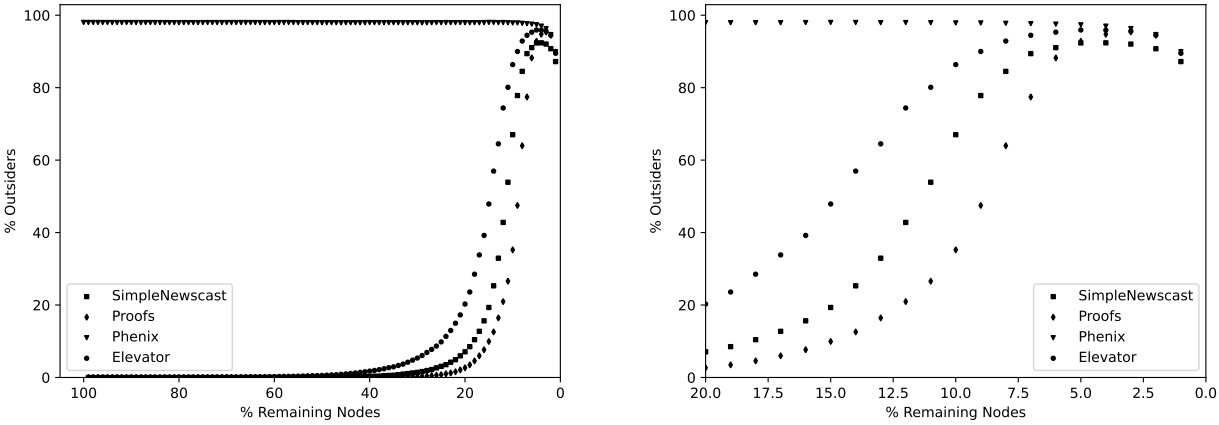
Fig. 17: Analysis of the removal of all the nodes of the network one by one, and observing the number of nodes outside the biggest strongly connected cluster, with churn.



(a) Removing of all the nodes

(b) Zoom on the last 20%

Fig. 18: Analysis of the removal of all the nodes of the network one by one, and observing the number of nodes outside the biggest weakly connected cluster, with a hub-targeted attack.



(a) Removing of all the nodes

(b) Zoom on the last 20%

Fig. 19: Analysis of the removal of all the nodes of the network one by one, and observing the number of nodes outside the biggest strongly connected cluster, with a hub-targeted attack.