Emergent Peer-to-Peer Multi-Hub Topology

Mohamed Amine Legheraba[1] Maria Potop-Butucaru[1] Sébastien Tixeuil[1,2] Serge Fdida[1]

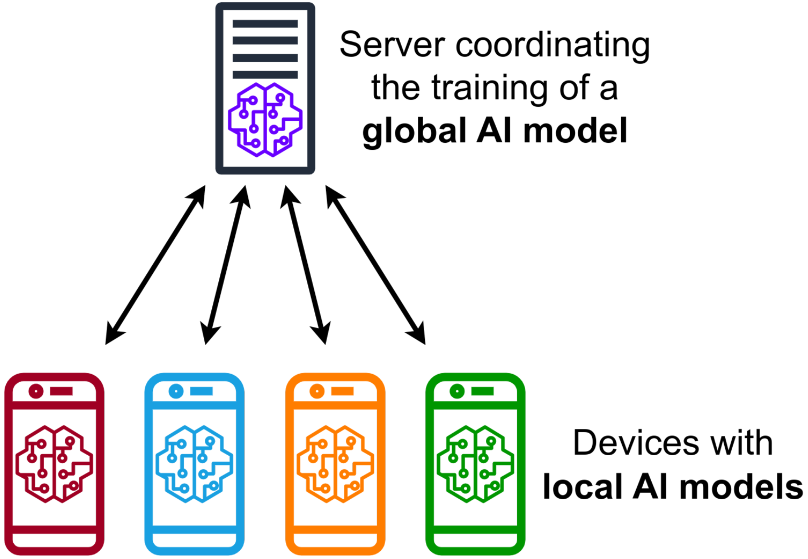
25 October 2024

[1]Sorbonne University

[2]IUF

Context

Federated Learning [1]



1

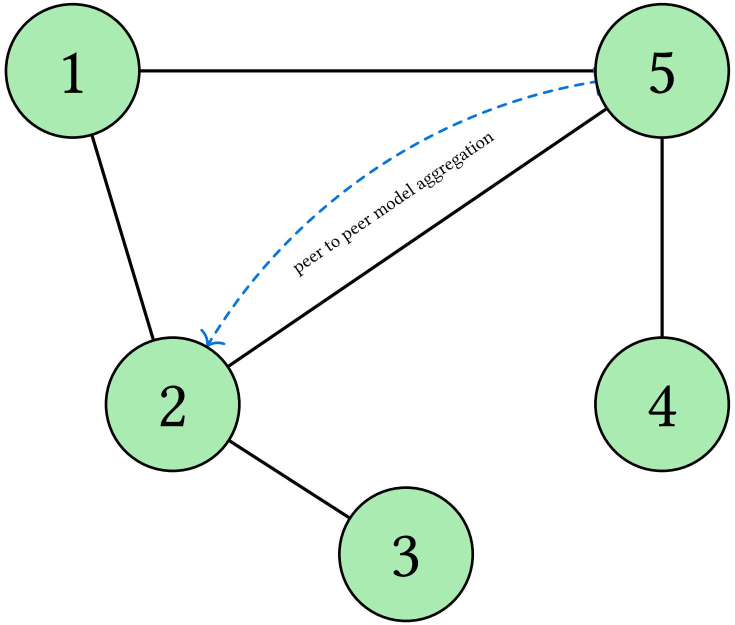
Algorithm 1 `FederatedAveraging`. The K clients are indexed by k; B is the local minibatch size, E is the number of local epochs, and η is the learning rate.

Server executes:
  initialize w_0
  for each round $t = 1, 2, \dots$ **do**
    $m \leftarrow \max(C \cdot K, 1)$
    $S_t \leftarrow$ (random set of m clients)
    for each client $k \in S_t$ **in parallel do**
      $w_{t+1}^k \leftarrow \text{ClientUpdate}(k, w_t)$
    $m_t \leftarrow \sum_{k \in S_t} n_k$
    $w_{t+1} \leftarrow \sum_{k \in S_t} \frac{n_k}{m_t} w_{t+1}^k$ *// Erratum*[4]

ClientUpdate(k, w)**:** *// Run on client k*
  $\mathcal{B} \leftarrow$ (split $\mathcal{P}_k$ into batches of size B)
  for each local epoch i from 1 to E **do**
    for batch $b \in \mathcal{B}$ **do**
      $w \leftarrow w - \eta \nabla \ell(w; b)$
  return w to server

[1]https://en.wikipedia.org/wiki/Federated_learning

Gossip Learning [2]



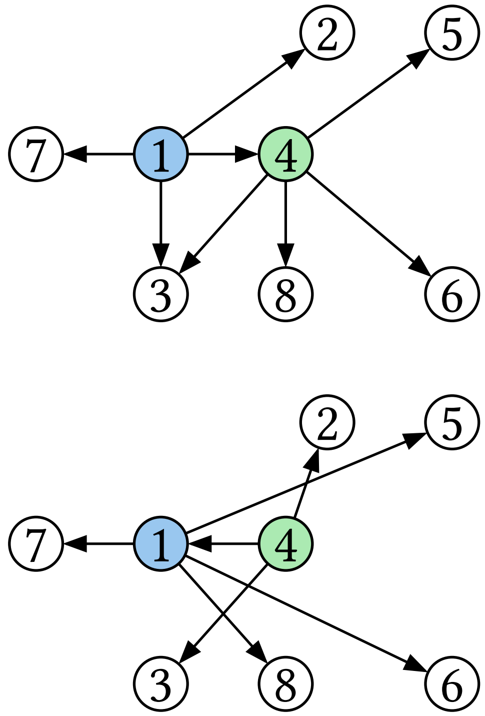
Algorithm 1 Gossip Learning Scheme

```
initModel()
loop
    wait(Δ)
    p ← selectPeer()
    send modelCache.freshest() to p
end loop
procedure ONRECEIVEMODEL(m)
    modelCache.add(createModel(m, lastModel))
    lastModel ← m
end procedure
```

Decentralized peer sampling [3]

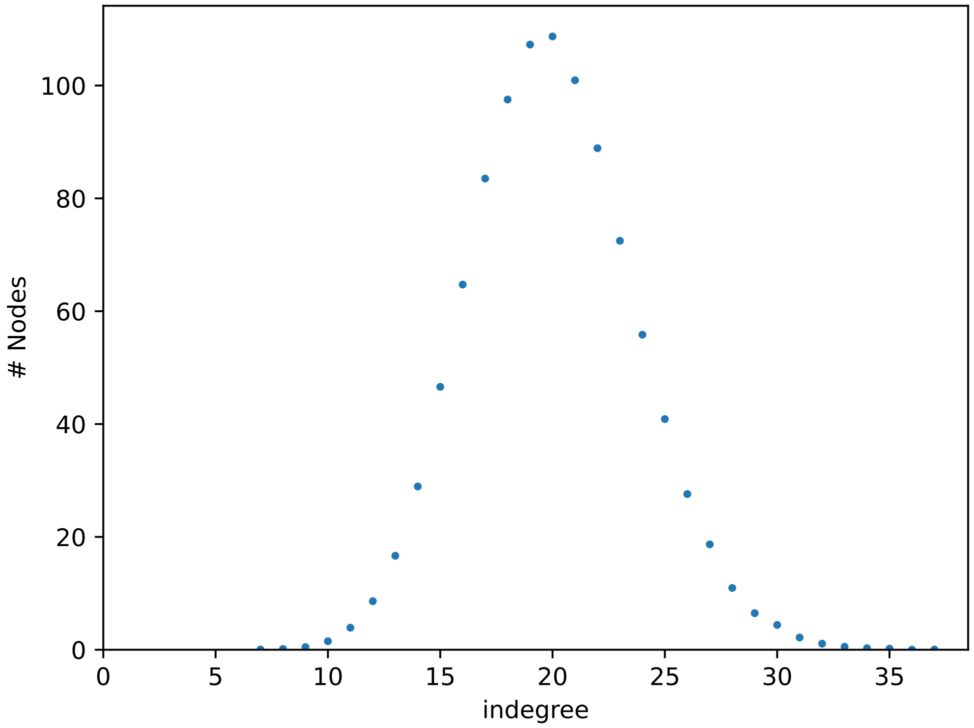
PROOFS Algorithm (active thread)

```
initial peer list: cache
cache size: c
shuffle length: l
node address: p
loop
    wait(Δ)
    subset ← selectRandomSubset(cache, l)
    q ← selectRandom(subset)
    subset.remove(q)
    subset.add(p)
    send(q, subset)
    subset_q ← receive(q)
    subset_q.remove(p)
    subset_q.removeAll(cache)
    cache ← subset_q
```

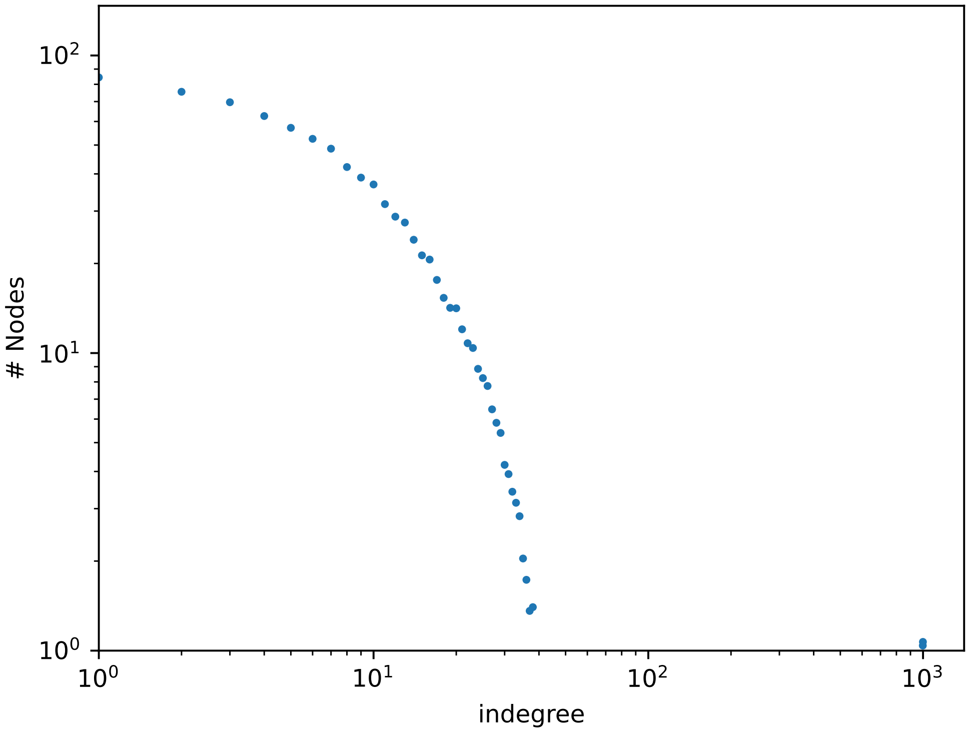


Before and after a shuffling operation. Node 1 sends addresses {itself, 2, 3} to node 4. Node 4 sends back {5,6, 8}.

Random-graph Topology [4]



Power-law Topology [5]



Elevator protocol

Hub-based topology & Hub sampling

- The objectif is to obtains an overlay network with **h** defined hubs, with **h** a parameter of the algorithm, and each hub is connected to all the nodes in the networks. The application running on top will be able to take advantage of this overlay to speed up message transmission in the network.
- **Preferential Attachment**: Drawing from the concept pioneered by Barabási and Albert [5], preferential attachment dictates that new connections in the network are established preferentially with nodes possessing a higher number of existing connections. 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 ([3], [6]), 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.

Algorithm

Elevator Algorithm (active thread)

```
initial peer list: cache
cache size: c
desired number of hubs: h
initial backward list: backward_peers (empty)
loop
    wait(Δ)
    frequency_map ← {}
    for peer in cache do
        peer_cache ← send(CACHE_REQUEST, peer)
        frequency_map ← frequency_map ∪ peer_cache
    preferred ← frequency_map.sort().select(c)
    preferred_backward ← {}
    for peer in preferred do
        peer_backward_peers ← send(BACKWARD_REQUEST, peer)
        preferred_backward ← preferred_backward ∪ peer_backward_peers
    cache ← {}
    cache ← selectRandom(preferred, h) + selectRandom(peer_backward_peers, c − h)
```

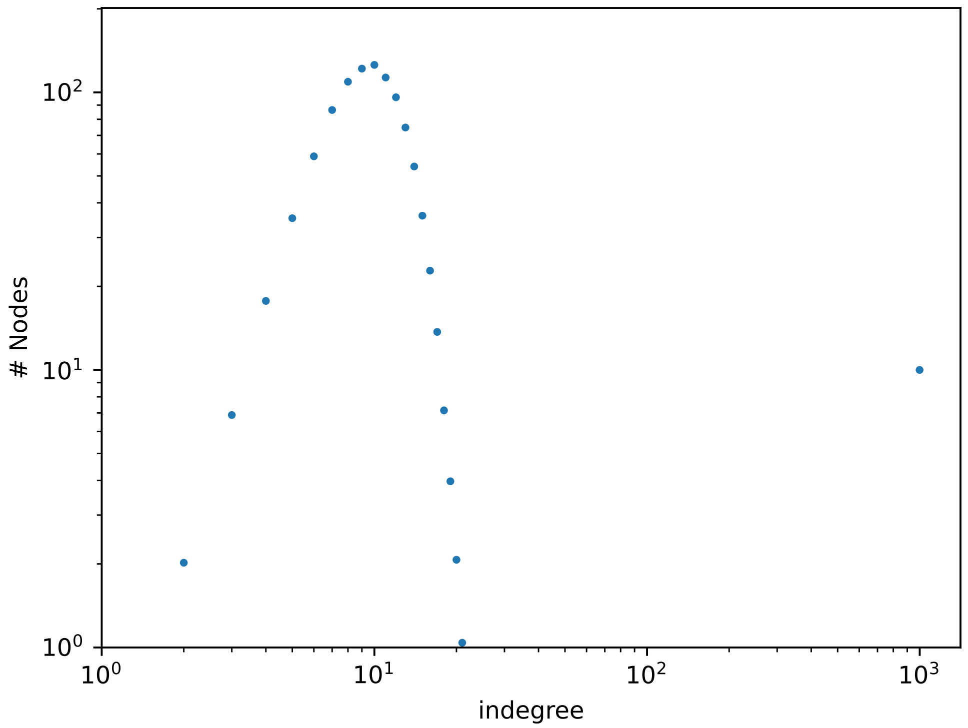
Simulation

- Simulations done with the Java PeerSim [7] simulator (modified), with the cycle based mode
- Comparaisons against 3 peer sampling algorithms : Newscast, Proofs and Phenix (Power-law)

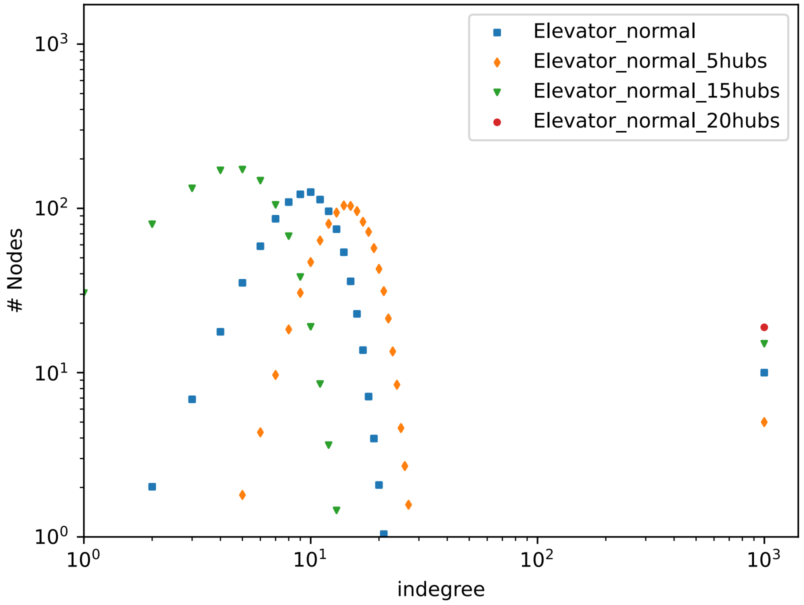
	Value
Network size	1000
Number of cycles for each simulation	1000
Number of times each simulation was run (with different seed)	100
c parameter (cache size)	20
h parameter (number of hubs)	10

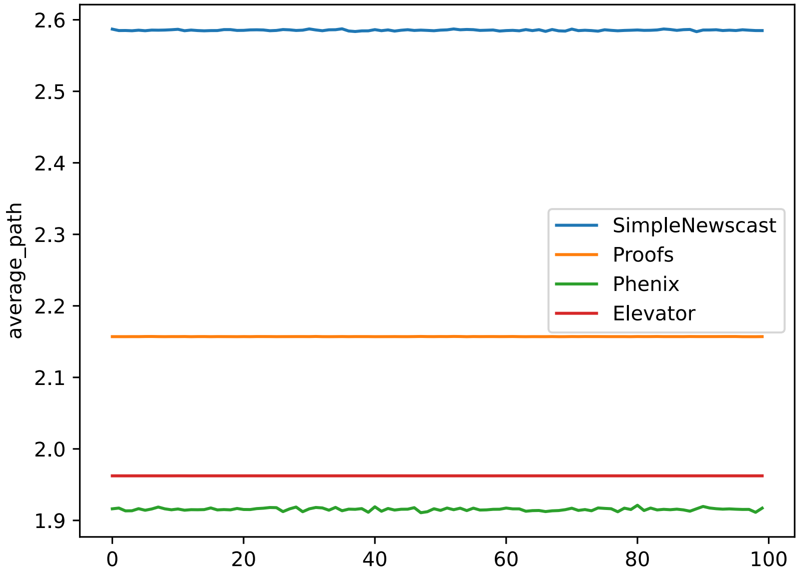
- All simulations were run on 16 vCPU, using 64G of memory.
- Code is available at https://gitlab.lip6.fr/legheraba/elevator

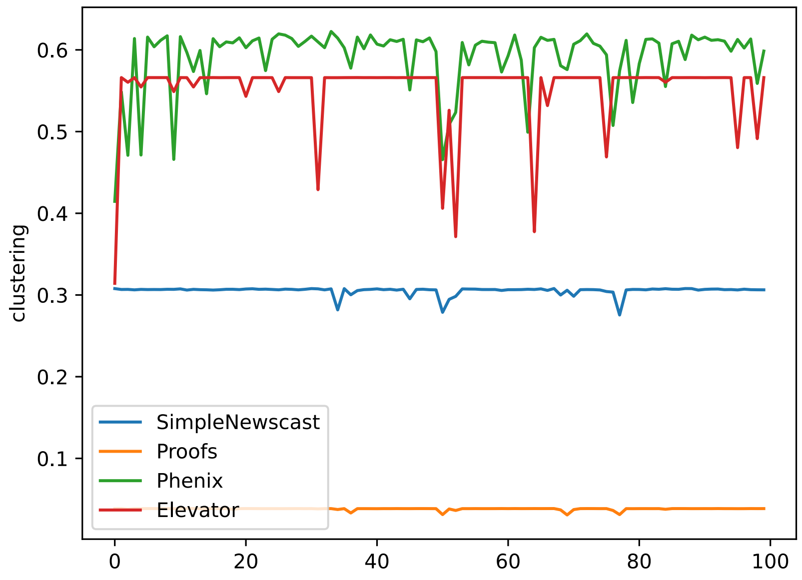
Distribution of indegree

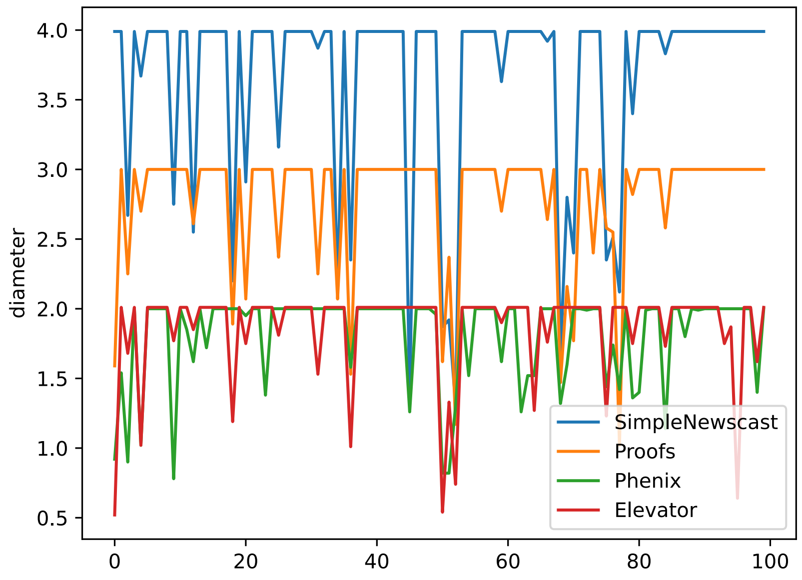


Metrics

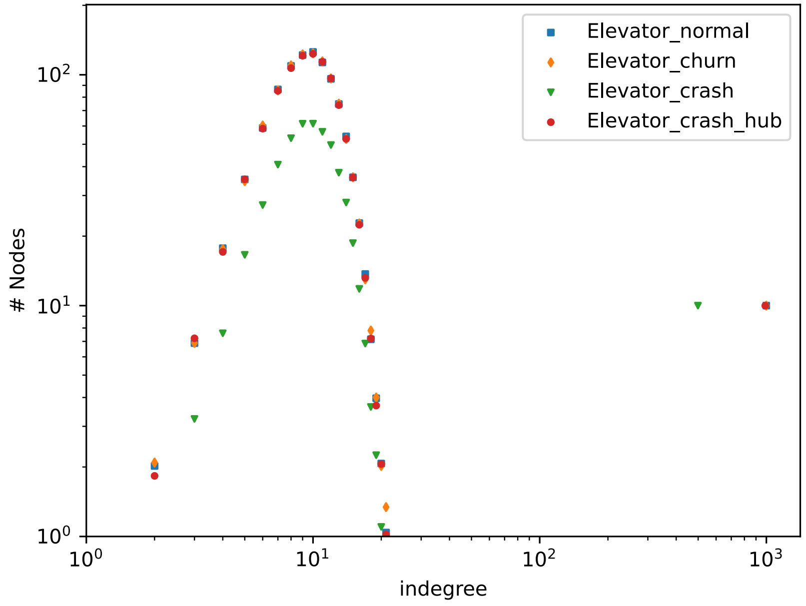

Elevator_normal
Elevator_normal_5hubs
Elevator_normal_15hubs
Elevator_normal_20hubs
Nodes
indegree
10^0
10^1
10^2
10^3

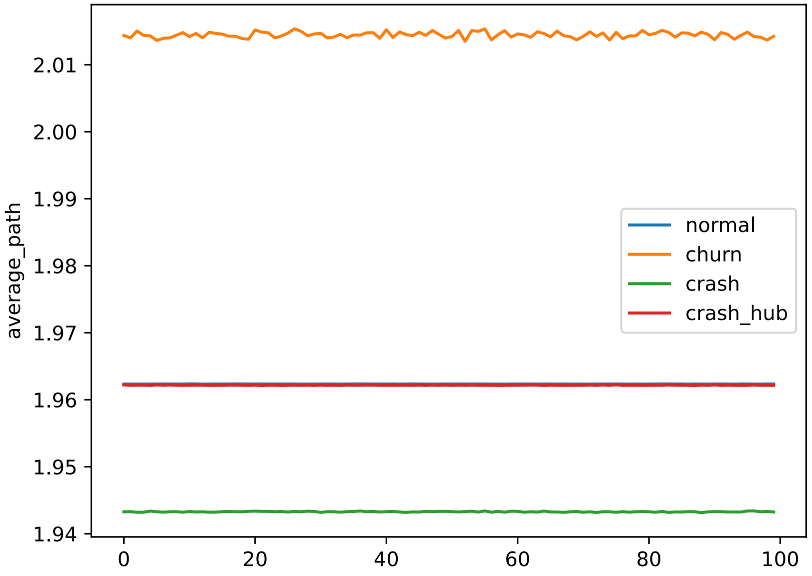

SimpleNewscast
Proofs
Phenix
Elevator
average_path
2.6
2.5
2.4
2.3
2.2
2.1
2.0
1.9
0
20
40
60
80
100

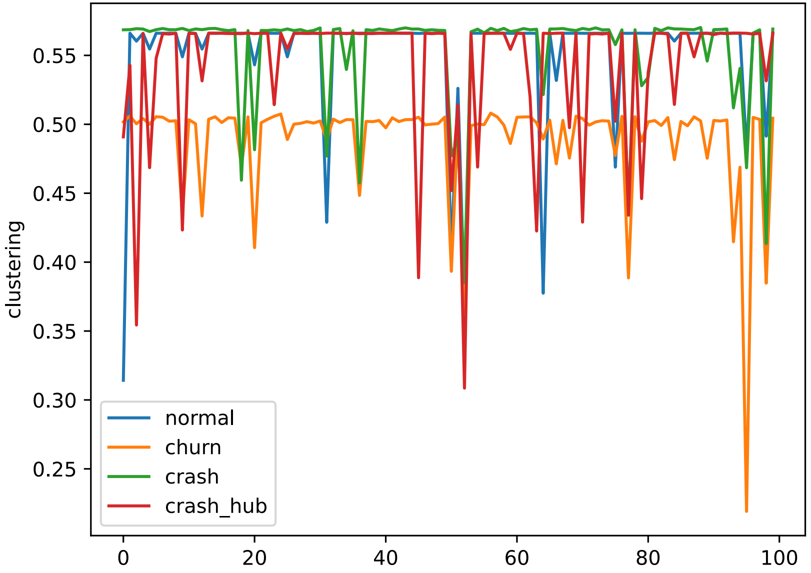

clustering
0.6
0.5
0.4
0.3
0.2
0.1
0
20
40
60
80
100
SimpleNewscast
Proofs
Phenix
Elevator

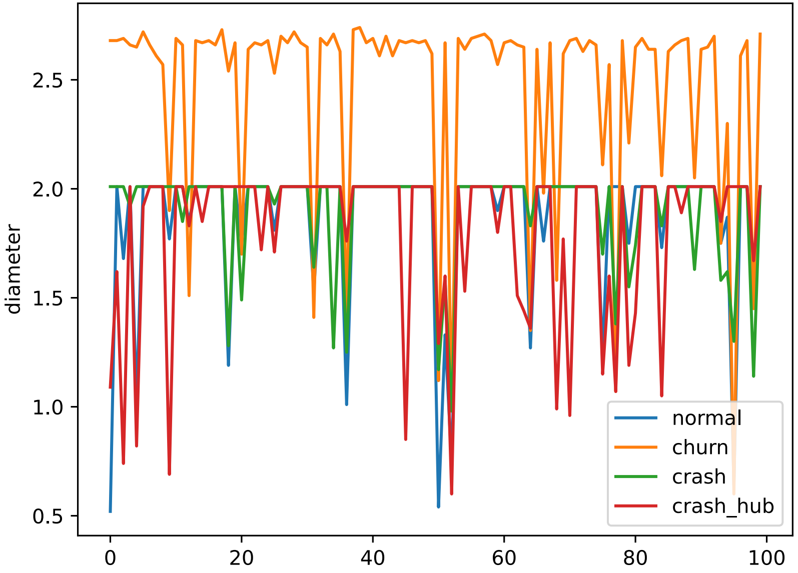

diameter
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0
20
40
60
80
100
SimpleNewscast
Proofs
Phenix
Elevator

Contexts


Elevator_normal
Elevator_churn
Elevator_crash
Elevator_crash_hub
Nodes
indegree
10^0
10^1
10^2
10^3


normal
churn
crash
crash_hub
average_path
2.01
2.00
1.99
1.98
1.97
1.96
1.95
1.94
0
20
40
60
80
100


clustering
0.55
0.50
0.45
0.40
0.35
0.30
0.25
0
20
40
60
80
100
normal
churn
crash
crash_hub


diameter
2.5
2.0
1.5
1.0
0.5
0
20
40
60
80
100
normal
churn
crash
crash_hub

Conclusion & next steps

Our algorithm allow the emergence of hubs and compared with PROOFS, Newscast and Phenix, our algorithm has equivalent results in term of shortest path length and diameter and is resilient against failures and attacks.

Next steps:

- Add machine learning & compare with other decentralized machine learning approaches ([2], [8])
- Adapt the algorithm to real physical networks
- Implement more complex attacks

Thanks

Any questions ?

Bibliography

[1] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, “Communication-efficient learning of deep networks from decentralized data,” in *Artificial intelligence and statistics*, 2017, pp. 1273–1282.

[2] R. Ormándi, I. Hegedűs, and M. Jelasity, “Gossip learning with linear models on fully distributed data,” *Concurrency and Computation: Practice and Experience*, vol. 25, no. 4, pp. 556–571, 2013.

[3] A. Stavrou, D. Rubenstein, and S. Sahu, “A lightweight, robust p2p system to handle flash crowds,” in *10th IEEE International Conference on Network Protocols, 2002. Proceedings.*, 2002, pp. 226–235.

[4] P. Erdos, A. Rényi, and others, “On the evolution of random graphs,” *Publ. math. inst. hung. acad. sci*, vol. 5, no. 1, pp. 17–60, 1960.

[5] A.-L. Barabâsi, H. Jeong, Z. Néda, E. Ravasz, A. Schubert, and T. Vicsek, “Evolution of the social network of scientific collaborations,” *Physica A: Statistical mechanics and its applications*, vol. 311, no. 3–4, pp. 590–614, 2002.

[6] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. Van Steen, "Gossip-based peer sampling," *ACM Transactions on Computer Systems (TOCS)*, vol. 25, no. 3, p. 8–es, 2007.

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

[8] M. De Vos, S. Farhadkhani, R. Guerraoui, A.-M. Kermarrec, R. Pires, and R. Sharma, "Epidemic Learning: Boosting Decentralized Learning with Randomized Communication," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

Appendix

Background thread

Elevator Algorithm (background thread)

```
loop
    request, peer ← receive()
    if request == CACHE_REQUEST then
        send(cache, peer)
        backward_peers.add(peer)
    if request == BACKWARD_REQUEST then
        send(backward_peers, peer)
```

Phenix algorithm

Phenix Algorithm (active thread)

```
peer)
    cache size : c
    initial peer list: cache
    initial backward list: backward_peers (empty)
    number of preferential connections: s
    loop
        wait(Δ)
        G_random, G_friend ← split(cache)
        cache ← {}
        cache.append(G_random)
        G_candidates ← {}
        for peer in G_friend do
            neighbor_list ← send(peer, CACHE_REQUEST)
            G_candidates ← G_candidates ∪ neighbor_list
        sort(G_candidates)
        G_preferred ← G_candidates[0..(s − 1)]
        for peer in G_preferred do
            send(peer, CONNEXION_REQUEST)
        cache.append(G_preferred)
```