Report on the peer sampling service simulation study for gossip-based distributed protocols

Matini Gabriele

1 The problem

Today gossip-based protocols are used everywhere in distributed systems, especially in large-scale ones, to exchange data. Peer-to-peer architectures in particular make extensive use of gossip-based protocols, and are known for their scalability. However, one underlying service any gossip-based protocol must rely on is the peer sampling service, which is the service that gives a peer a partial view of, and a way to communicate with the overall network. In particular, a study was needed on the performance and dependability of such service with respect to how it is implemented. Also, it was often (wrongly) assumed that such service would induce an overlay network with characteristics similar to those of a random overlay network. As we will see, this is not always the case. In this research the solutions presented in the paper of reference were implemented in order to test the paper's claims and recreate the experiments.

2 The technologies

The simulation was programmed with Simpy, a useful python library which can simulate distributed environments. In particular, Simpy utilizes a communication channels system, in which each channel acts as a mailbox, in which messages are deposited and recovered by the simulation environment entities. Additionally, the simulated network has been abstracted into a graph thanks to the Networkx library, which also helped in collecting data over the network. Finally, Matplotlib was used to plot said data in an understandable way.

3 The implementation

3.1 The overall scheme

The implementation tries as much as possible to follow the experiments reported on the reference paper: in particular, the two main case studies present in the paper were implemented, those being the growing overlay case study and the random overlay case study. The peer sampling service protocols were divided among three different functions to implement, each of which have three

different implementation alternatives: peer selection, view selection and view propagation. First of all, a node is equipped with a fixed size partial view of the entire overlay network, and this can be implemented as a list of (peer_address, hop_count) tuples ordered by hop count, that take the name of descriptors. The peer selection policy directly affects how a peer is selected from the view, so we can either select a random peer, the peer with the lowest hop count or "head" of the list, or the peer with the highest hop count, or "tail" of the list. So a peer A selects a random peer B from his view, merges this view with his own descriptor (with hop count 0), and sends this view to B to inform B of the existence of A and all the nodes in A's view. However, B has a fixed size view (let this size be k), so in doing the update B needs to choose exactly what information to keep from A and what to discard: again, this can be done by either selecting randomly the descriptors to keep, selecting the first k descriptors (the first k nodes with lowest hop count, or "head"), or selecting the last k descriptors (k nodes with highest hop count, or "tail"). This function is called view selection. The third policy regulates how the information is exchanged: peer A might probe peer B to get B's view ("pull" policy), or just send B his own view ("push" policy), or send his own view to B and B will send his view back to A as an answer ("push-pull policy"). The different choices for these functions amount to 27 total combinations, which we will refer to with the triple (peer selection policy, view selection policy, view propagation policy). The whole simulation takes place on a workload of 100 nodes in total, and looks at the evolution of the overlay network over 300 cycles or simulated time units. The reduction of the workload was due to the limitations of the technologies used, however, as we will see, the quality of the simulation was marginally affected and with 100 nodes we are also able to provide comprehensive pictures of the network evolution. The view size was reduced from 30 to 4: having each node have a view that amounted to 30% of the whole graph would have skewed the results. At the same time, 4 was the smallest number possible such that some protocols would not suffer from partitions (see later). Finally, a dependability study was conducted in a similar way to the paper of reference, evaluating failure tolerance and self-healing capacity of the network.

3.2 The parameters

In our performance evaluation three main parameters were of interest: the node degree, the average path length and the clustering coefficient. The dependability evaluation will thus move along this three important axes. The node degree was examined along the paths of average node degree and evolution of the node degree distribution in the network. The degree distribution is important to evaluate tolerance to failures, as well as the speed of the any gossip-based protocol, and to determine if there are communication hotspots in the overlay. The average path length is the average over all shortest path lengths between all pairs of nodes in the graph and it is performance wise important, since an u-v shortest path defines the lower bound of the cost of communication between node u and node v in the overlay. The clustering coefficient of the overlay network is

defined as the average of the clustering coefficients of the nodes, which are in turn defined as the division between the number of edges between the neighbours of a node and all the possible edges between those neighbours. A high clustering coefficient is undesirable, since it multiplies the number of redundant messages exchanged in any gossip-based protocol, and it weakens the cohesion between a cluster and the overlay, thereby increasing the probability of partitioning. The clustering coefficient is a number between 0 and 1. It is important to note that connectivity is of paramount importance to calculate these parameters, and that we are not interested in partitioned networks anyway, so in the simulation only non-partitioned networks were analyzed. Secondly, it is worth to point out that out of the 27 possible combinations, only 8 will be analyzed. In fact, (head, *, *) protocols result in severe clustering: as shown in the "High Clustering Proof" graph, clustering for these protocols amounts to 0.7 or higher which is much higher than any of the protocols we will see. At the same time, (*, tail, *) protocols cannot handle joining nodes well, since as shown in the related proof, the graph converges to a structured topology centered around a few hotspot nodes, thus resulting in new nodes having to compete with the majority of the other nodes in order to communicate with anybody. Finally, any (*, *, pull) policy will result in a star topology, which is highly undesirable because of low tolerance to failures (if one of the central nodes crashes the probability of partitioning is high), and low scalability.

4 Dependability evaluation

4.1 Growing overlay case study

In the first scenario, we have one node at time 0, which is the central node any joining node will use in order to bootstrap itself into the network. A new node will join at every cycle until the network reaches 100 nodes. New nodes are thus initialized with the descriptor of node 1 in their view. This case study was designed to show how the protocols perform on the worst bootstrapping case scenario. The paper studies 6 out of the 8 protocols on this scenario, since two protocols are highly susceptible of partitioning, reporting a much faster convergence to some kind of random graph if push-pull is used rather than push. Our simulation shows this too: This result can be easily explained by the

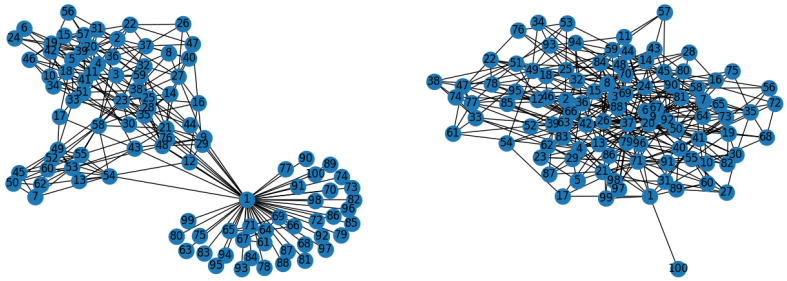

Figure 1: To the left: (rand, rand, push) implementation at cycle 100. To the right: (rand, rand, push-pull) impl. at cycle 100. We can clearly see how node 1 is still a bottleneck for the overlay on the left, while the graph has already converged to an apparently random topology on the right.

fact that the push-pull policy involves an overall faster exchange of information between the nodes.

4.2 Random starting graph case study

In this scenario we start from time 0 of the simulation with 100 nodes and a completely random overlay. One thing to note is that, comparatively with the growing overlay scenario, the protocols converge to more or less the same values in degree distribution and average path length. This can be proven by just running a simulation for case study 1 and one for case study 2 with the same protocol. At the same time, results of simulation runs are already provided and can be examined for a confrontation. In terms of clustering coefficient, average path length and average node degree, (*, rand, push-pull) are the best with the highest average node degree consistently around 7.5, the lowest shortest path length around 2.5 and lowest clustering coefficient around 0.2, with (tail, rand, push-pull) being slightly better than (rand, rand, push-pull) in all three categories. These better approximate a random graph. They are followed by

the (*, rand, push) protocols with an average node degree around 6.75, average shortest path length of around 2.7, and a clustering coefficient around 0.3-0.35 for both protocols. There are in turn followed by the (tail, head, push) protocol, having average node degree around 5.8, average shortest path length around 3.3, and clustering coefficient slightly above 0.35. Then we have (rand, head, push), with average node degree around 4.2, average shortest path length of almost 4, but clustering coefficient consistently around 0.225. Finally, we have the (*, head, push-pull) protocols with clustering coefficient bigger or equal than 0.35, an average node degree of 4.2 or below and average shortest path length of 4.5 or higher. Note that, generally speaking, the results are consistent with the paper graphs for random overlays in Fig.3: the values are not exactly the same due to the sample graph being different, however the trends are consistent.

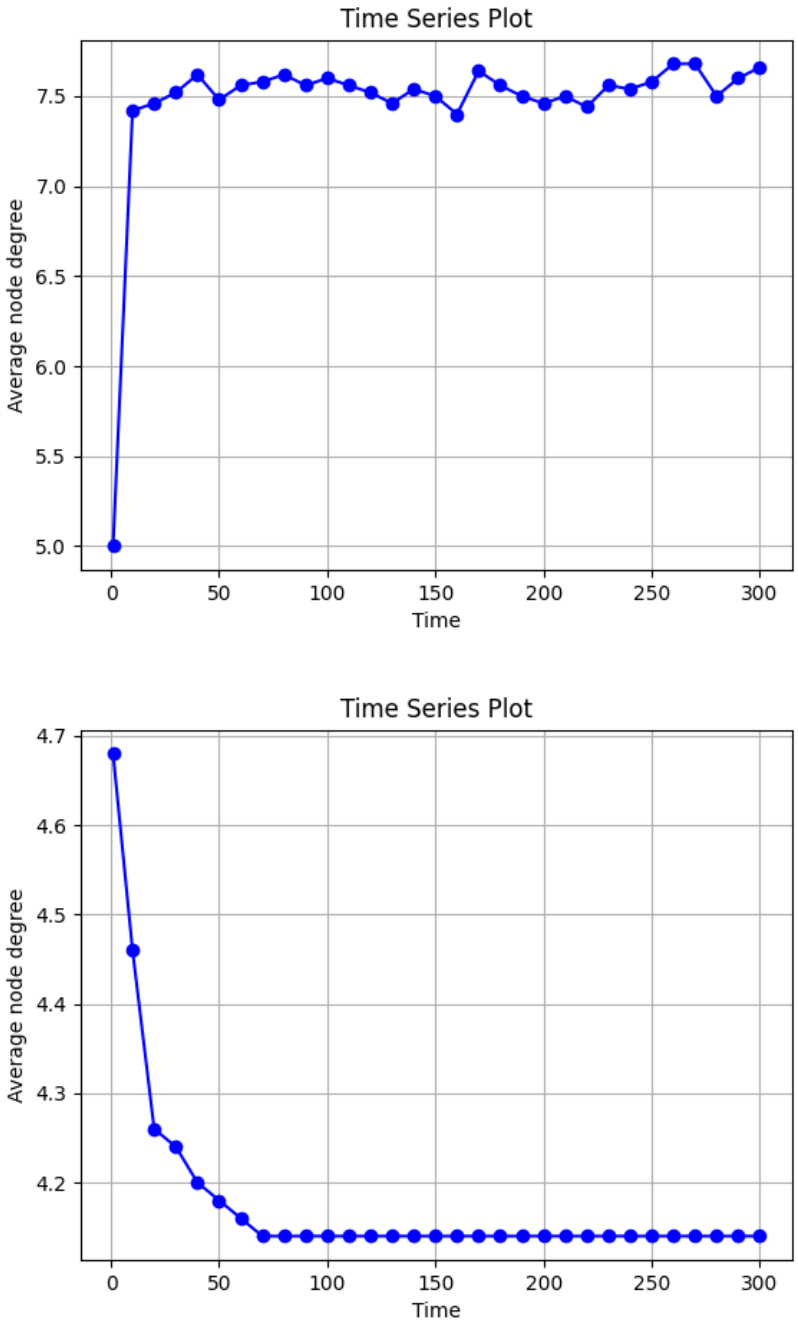

Figure 2: Average node degree comparison between one of the "best" protocols, (tail, rand, push-pull) with one of the "worst" (rand, head, push-pull). As we can see, the protocols start from similar values, but stabilize at very different values on convergence.

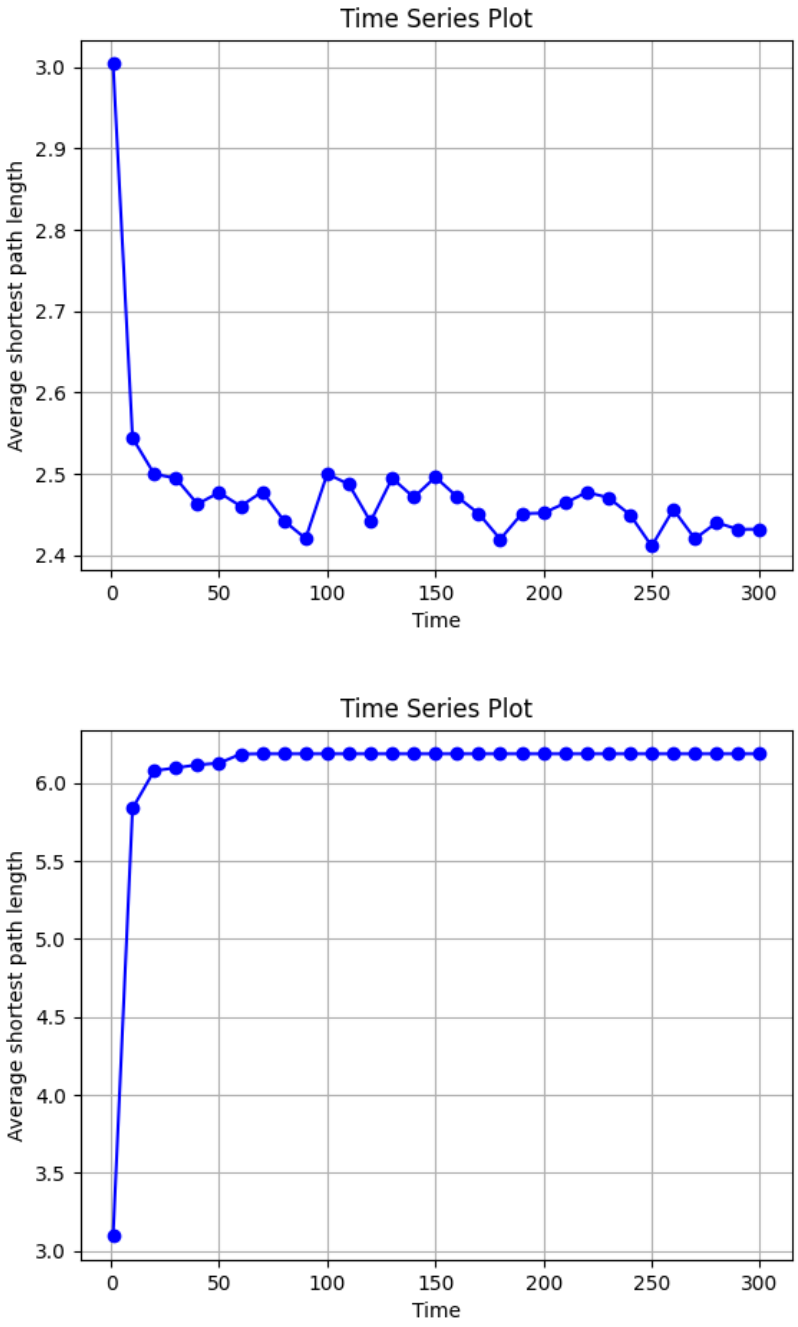

Figure 3: Average path length comparison between one of the "best" protocols, (tail, rand, push-pull) with one of the "worst" (rand, head, push-pull). As we can see, the protocols start from similar values, but stabilize at very different values on convergence.

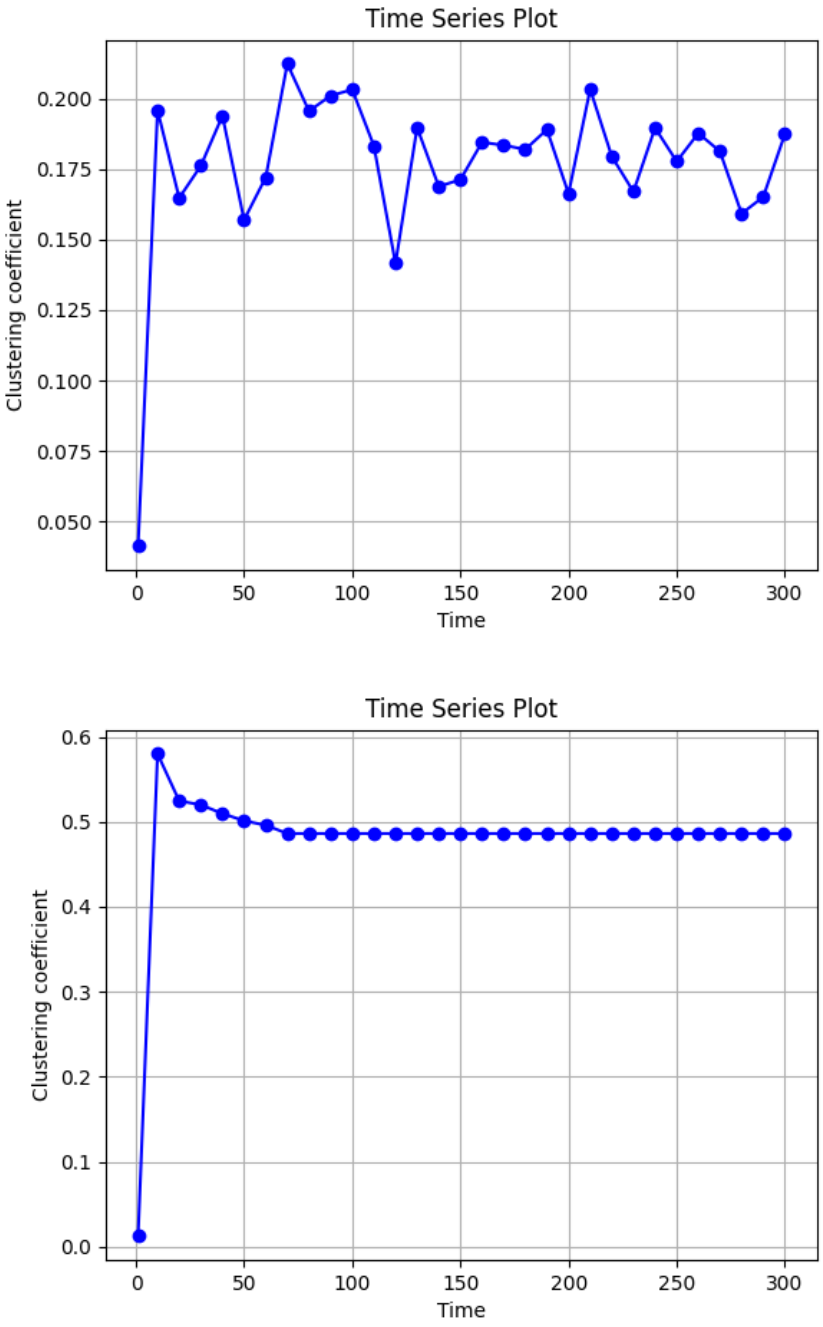

Figure 4: Clustering coefficient comparison between one of the "best" protocols, (tail, rand, push-pull) with one of the "worst" (rand, head, push-pull).

4.3 Node degree distribution

The paper follows with a degree distribution study focused on its evolution in section 6. 50 nodes were studied. In our simulation, we calculated the mean over the mean degree of the nodes in 300 cycles, so for each node i, let d(i, j) be the degree of node i in cycle j: we calculate di as the mean degree of node i over all 300 cycles and d = (d1 + d2 + ... + d50)/50. The we calculated out of these values variance and standard deviation as reported on the paper. The results were interesting: all in all, the protocols can be divided over (*, head, *) and (*, rand, *) policies, as (*, head, *) seem to converge faster with lower variance and standard deviation (below 1) throughout the process and lower d (around 4.5). At the same time, (*, rand, *) protocols have higher values in variance, std deviation (1 or above) and higher d value (around 7.5). These values can be confronted in the provided txt files "statistical values.txt". The results are line with the trends reported on the paper, however (rand, rand, push-pull) and (tail, head, push) seem to deviate from the overall trend in variance. This is probably due to the fact that the network and the views are smaller, so these protocols have less opportunity to show their true characteristics due to an overall fast convergence. In general, variance and standard deviation tell us how evenly the degrees are spread throughout the cycles, for all the nodes: lower variance means that in general the graph does not concentrate around a few nodes. In the case of random view selection the network is less "stable", in this sense. However, it is important to note that at the end all nodes will converge to the same degree with no node having a significantly higher degree than the rest. To end the study on degree distribution, we are finally interested in determining the randomness of degree distribution. Thus, we calculated and plotted the auto-correlation coefficient of the degree values of a node separated by k cycles, for every possible k. The paper studied four protocols, again, it seems to be possible to divide them into two camps: (*, head, *) protocols appear to be more random, with the coefficient going almost instantly to zero compared to the (*, random, *). It is important to note that sometimes the auto-correlation coefficient results will vary a lot, with the trend resembling a linear trend. This may be due to the node finding itself in a strange starting situation or due to the conformation of the starting random graph. However, most of the time the results will be consistent.

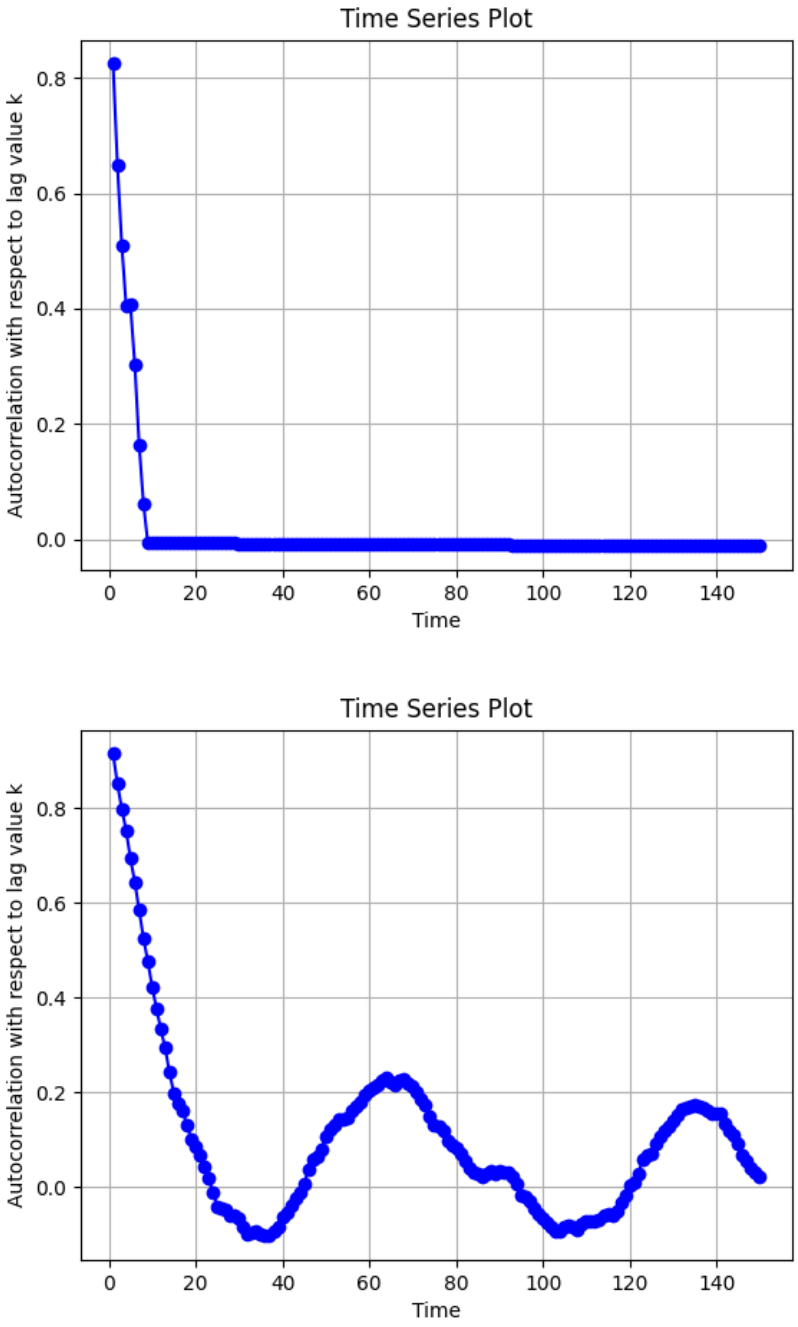

Figure 5: Auto-correlation between (rand, head, push) above and (rand, rand, push) below.

4.4 Tolerance to failure

To end, we present the study about failure tolerance. The study will be divided into two core areas: a static area and a dynamic one. For the static area, the paper focuses on counting how many nodes on average are left out from the main cluster in a given protocol. However, it was considered more appropriate to study the probability of having partitions at all, given a percentage of nodes removed. So what we do is stop the simulation at time 300, and remove nodes progressively from the graph until we create a partition. The experiment will be reproduced 800 times to get an average on the percentage values of removed nodes from the graph. Doing these many experiment will allows us to get a good idea of the probability of having a partition if k percent nodes are removed from the graph. We will observe that, in fact, the protocols that result in the highest clustering coefficients and lowest average degree are the most susceptible to partitions: (*, rand, push-pull) are the most tolerant, with the probability of having a partition getting near 1 if 70% of nodes or more are removed. Next we have the (*,rand,push) protocols that can tolerate up to around 65% of nodes removed. Then (tail, head, push) and (rand, head, push) follow with around 60% and 45% respectively. The least tolerant protocols are (rand, head, push) and (rand, head, push-pull) with 40% and 25% respectively.

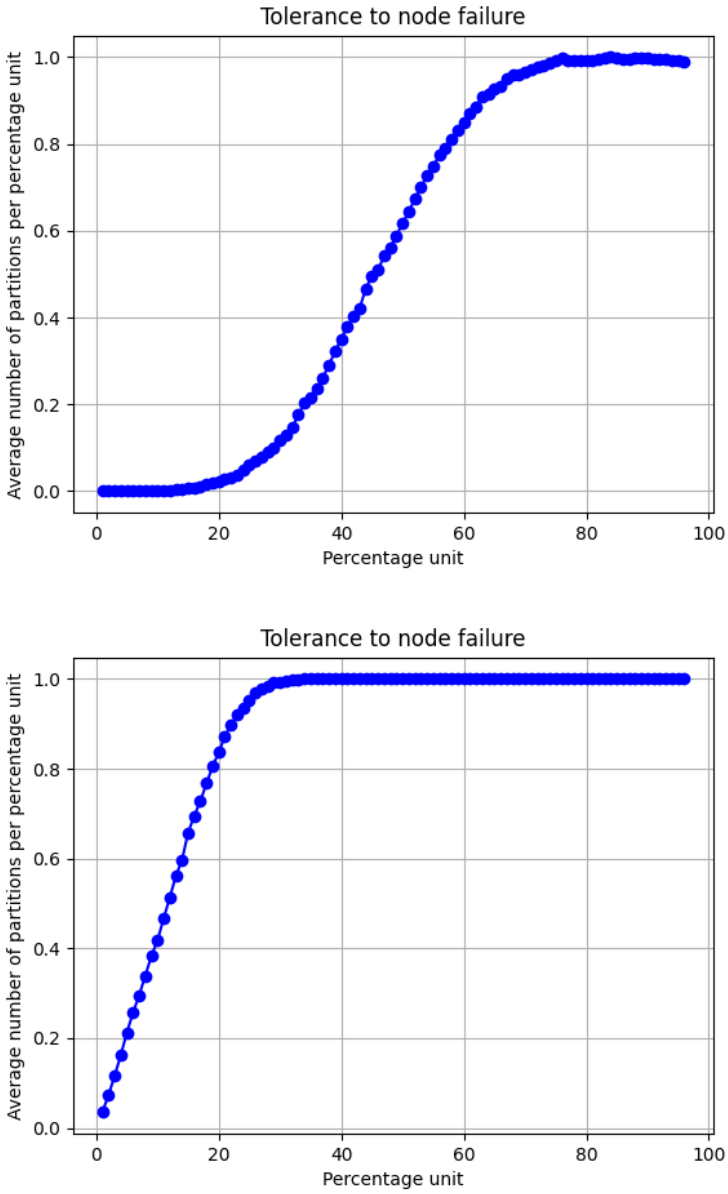

Figure 6: Comparison of tolerance between one of the most tolerant protocols (random, random, push-pull) and one of the least tolerant (random, head, push-pull). As we can see, the probability of partitioning approaches 1 as soon as we remove around 25% of the nodes from the graph for (random, head, push), while we have to remove almost 70% of the nodes to be sure to get a partition.

4.5 Self-Healing capacity study

The last study focuses on the behavior of the protocols in presence of massive node failures. In particular, it is expected that the protocols regain some kind of connection and reconfigure the graph if no partitions were created. Thus, the study takes the graph at cycle 300, removes 50% of the nodes, paying attention to not create any partitions, and measures the amount of dead links (missing links) from all the nodes' views and how they vary with time until cycle 350. At the same time, the view size has been increased from 4 to 8: the reason for this decision was to be able to remove 50% of nodes and be able to study even the less tolerant protocols, since a larger view helps with the failure tolerance as shown in the related proof graph. In this way we are sure to be able to remove 50% of the nodes and cause no partitions for every protocol. In this case study the graph was admittedly almost too small, as many protocols converge very rapidly. However, we could still trace an overall pattern of (*, head, *) protocols being significantly faster in self-healing than (*, random, *) protocols. However, it is worth to note that results may vary, most probably because of the small sample compared to the one of 10000 nodes used by the paper. In particular there are striking differences between (rand, head, push) and (tail, head, push): in the paper it was recognized that (rand, head, push) is significantly faster than (tail, head, push) and here this difference makes it so (rand, head, push) almost instantly converges to 0 dead links, which makes it difficult to compare with other protocols.

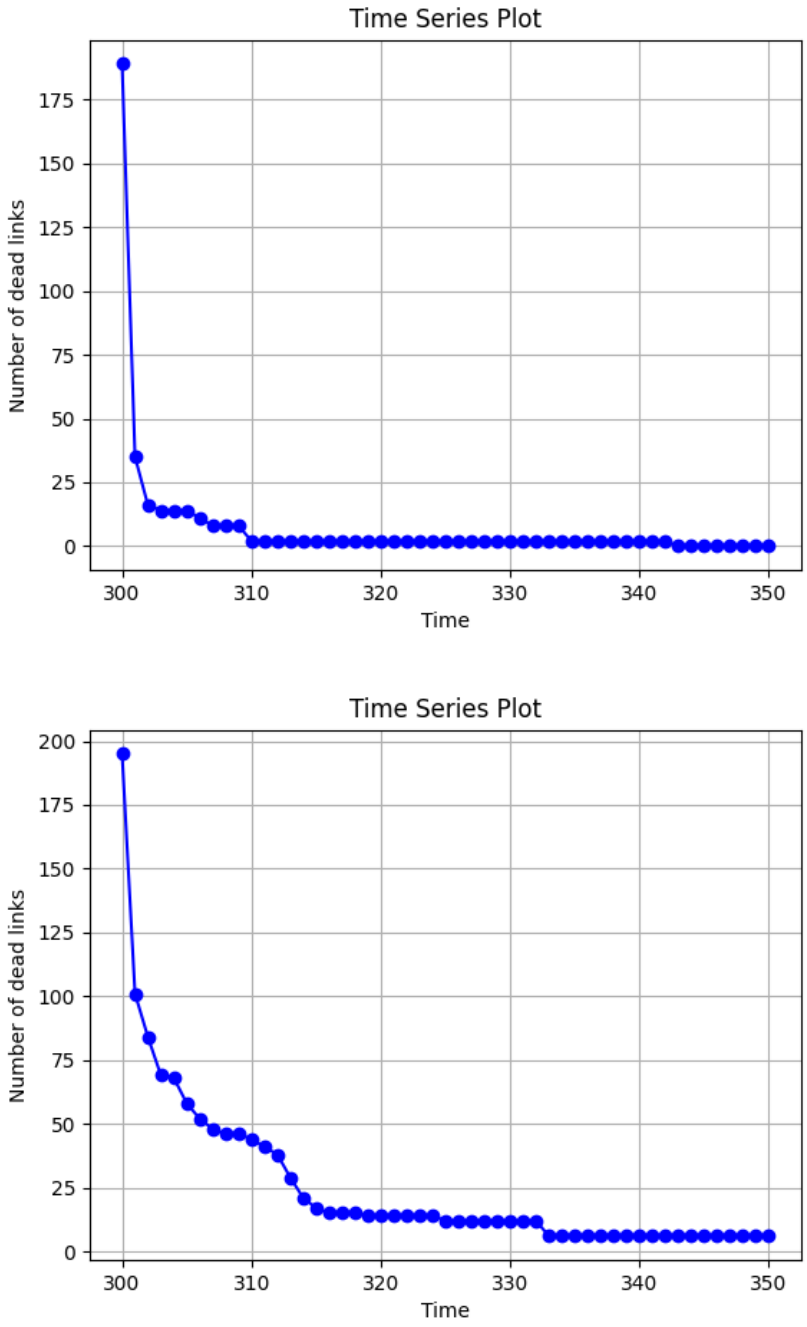

Figure 7: Healing comparison between (tail, head, push) above and (tail, rand, push) below.

5 Installation and usage instructions

The needed dependencies are the python libraries are random, networkx, copy, matplotlib and simpy. For the installation, it is enough to download the files from this repository:

`https://github.com/kev187038/PeerSampleServiceForGossipBasedProtocolsSimulation`

To run the simulation just type inside the folder

```
python3 simulation.py
```

The simulator will first ask to set the scenario (growing overlay or random graph). If the random graph scenario is chosen, the user will we prompted for abilitating the dynamic study or not. Choose yes only if you are interested in the self-healing case study, since it is treated as a completely different study from the rest. Next the user will be prompted to choose the protocol: the first choice is the peer selection, which can be "random", "head", or "tail"; the second is the view selection with the same options. Next we have the option to abilitate push-pull mechanisms to activate push-pull instead of having only push. In the end, the user will also be asked if they want to plot the graph in real time or not: if the user chooses "yes", they will be prompted for the lag time which can be a value from 1 to 100. The lag time expresses how much cycle have to pass before the simulation shows the user the graph in the current cycle. So, if the user chooses 100, the simulation will show the graph at cycle 1, then at cycle 100, then at cycle 200 and finally at cycle 300. Values of 10, 30, 50 or 100 are recommended. At the same time, the lag value also specifies at what time the measurements for clustering coefficient, average path length and average node degree are taken. This will influence the final form of the data charts. Other parameters such as auto-correlation, variance or static fault tolerance are calculated separately and only in the random graph case study. Data charts are already available for all cases in order to not have to run the simulation and just take a look at the data.