UNIVERSITY OF TRENTO

Project of Distributed Algorithms

IMPLEMENTATION OF CYCLON OVER THE AKKA FRAMEWORK

Luca Zamboni
XXXXXX

Luca Erculiani
XXXXXX

Abstract

The topic of this work is the implementation of Cyclon, a decentralized peer-to-peer protocol for gossiping over the Akka framework.[1] The goal of Cyclon is to build a network that can resist against crash of a great part of its node without collapsing in a series of disconnected clusters. This document will explain first the theoretical basis of the protocol, then our implementation of a program pacable of simulating a Cyclon network. The last part of this work will be focused on the statistical result of simulations made, discussing the capabilities of the network to build a strongly connected graph, to resist to massive disconnection. The last test will show the limits of the protocol in case of hub attack.

1 INTRODUCTION

The goal of this work is to implement a simulation of a peer-to-peer network running the Cyclon gossiping's protocol, for the project of Distributed Algorithms, by Alberto Montresor. The network is strongly decentralized and consist in a large number of peer clients and one or more tracker server, whose function is to allow new peers to connect into the network. Now will follow a list of fast description of the topics of every section of this document.

- The system model section will describe the Cyclon protocol, showing the general structure of the algorithms that implement it and focusing on the particular elements that characterize it.
- The implementation section is focused on explaining the actual implementation of our simulation, discussing about the choices made by development and the input and output of the program.
- The analysis and result section will show information about tests executed on the program, for evaluating some characteristic of Cyclon

2 SYSTEM MODEL

The system that we want to model is a dynamic collection of distributed nodes that wants to participate in a epidemic protocol. The number of node isn't fixed and can increase or decrease depending on peers who join and leave the protocol, or maybe crash. The communication between pairs of nodes needs that one of them knows the address of the other one, and the channel is a best-effort type (potentially a lot of message omission).

2.1 SERVICE SPECIFICATION

Nodes has only a partial views of the network, and this view is dynamic. Each node periodically gossip with a random neighbour about its other neighbour. The main idea is that nodes continuously exchange information about other nodes, removing the old ones (so the most probable disappeared) and adding the new ones. Each node has a fixed number of neighbour and shuffle this with other nodes. For communicating with another node, a peer needs a neighbor descriptor of that node, consisting:

- the address of that node
- a timestamp information about the age of the descriptor
- more additional information, maybe needed by the upper software layer[2]

When a new node want to enter in the protocol asks to a tracker server a random subset of peers and start the communication.

2.2 SKELETON OF THE ALGORITHM

Here is presented the structure of the code of an instance running Cyclon. The first algorithm shown is common to a bunch of protocol with the same purpose, like Newscast.[3]

```
**upon** *inizialization* **do**
    $view \leftarrow$ descriptor(s) of nodes already in the system
**repeat** *every* $\Delta$ *time units*
    Process $q \leftarrow$ selectNeighbor($view$)
    $m \leftarrow$ prepareRequest($view, q$)
    **send** $\langle$REQUEST$, m, p\rangle$ **to** *q*
**upon receive** $\langle$*REQUEST*$, m, q\rangle$ **do**
    $m \leftarrow$ prepareReplay($view, q$)
    **send** $\langle$REPLY$, m', p\rangle$ **to** *q*
    $view \leftarrow$ merge($view, m, q$)
**upon receive** $\langle$*REPLY*$, m, q\rangle$ **do**
    $view \leftarrow$ merge($view, m, q$)
```

What make every protocol different is the behaviour of the three functions called in the previous pseudocode. In Cyclon these components acts in this way:

- selectNeighbor() selects the oldest neighbor in the view
- prepareRequest($view, q$) removes $t-1$ random descriptors from the view and return this subset plus a fresh local one
- prepareReply($view, q$) removes and return t freshest neighbors from the local view
- merge($view, m, q$) merges the local view and the one received, if there are duplicates keeps only the freshest. Remove itself and reinsert entries sent to q if space permits

3 IMPLEMENTATION

We have implemented protocol Cyclon with Akka framework in Java. We choose Akka because it offers a really good scalable Peer simulator and easy manage interface messages exchanging.

3.1 MESSAGE

There are two type of message in our Project. When a Peer is created it sends a message of type Message.java to the tracker it contains only the information about the sender and nothing else. The other message of type MessagePeer.java contains the sender of the message and the list of Peer that you are exchanging and also the type of the message. MessagePeer.java can take different values:

- 0 If is the response of the Tracker
- 1 If a Peer start an exchange sending some Peer to another Peer.
- 2 If it is a response for another that sent some Peer
- -1 If this Peer is Died. This is only to simulate an under layer that says if a Peer is died.

3.2 TRACKER

Tracker.java has the task to give some address of Peers to a Peer that has been created. It receive a Message of type Message.java from the Peer, it subscribe the new Peer to the local List of Peers in the network. After that it choose n newest Peers in the list and send it back to the Peer a message that contains the list of Peer.

3.3 PEER

This is the main class of the project. It simulates a role in the network of Peers. This class has the following structure.

- preStart(). It basically request the peer from tracker sending a Message.java and subscribe this peer to the monitor only for get data for statistic.
- onReceive(Object message). This function handles messages. If type is 0 is just in itialize the list of neighbors. If type is 1 it sends a message of type 2 with his n neighbors plus him to the sender of the message. After it merge the received Peers. If type is 2 it just merge the received Peer from the sender. If the type of the message is -1 it remove the sender from his neighbors.
- merge(int type, ArrayList<Peer> peers). Is a function that merge peers with local neighbors and if the size of the list remain under the minimum size it fill with older neighbors known in the previous cycle.
- send(int type, Peer to). If the type of the send is 1 it choose n-1 random peer. If it is 2 it choose last n peer with higher timestamp. After this it sends the chosen peer to the Peer.
- selectPeerToContact() select the neighbor with lower timestamp.
- run() After you get a reply with peers it waits DELTA time to restart a cycle.
- removeNodeFromNeighbors(Peer peer). Remove peer from list neighbors.

3.4 COLLECTING DATA

When a Peer is initialized he subscribe itself to the monitor. The monitor every DELTA seconds write on file neighbors of all peer with the number of average cycle of the list of peer and start a GUI that shows the evolution of the neighbors of peer.

3.5 SIMULATION

When a Peer is created it subscribe itself to the monitor just to collect data for statistics. After it send a request the initial list of Peer to the Tracker. The Tracker write it to the global list of peers in the system and send back the subset of the newest in the network. The peer now start first cycle of his life. It select the peer with older timestamp and sends him his n-1 random neighbors with him added to the list with a fresh timestamp. The peer that receive the message select his older n neighbors and send it back to the sender and merge those who arrived with its local neighbors. Now the peer that started the request receive the list of peer and merge it with is's local view of the network and after that it start a new run after DELTA time. Every n second monitor write on file the list of neighbors of each peer.

4 ANALYSIS AND RESULTS

In this section we will discuss the results of simulations over the implemented protocol and some tests done under partarticular conditions.

4.1 ANALYSIS METHODS

The output of the program was used to build graphs, using the NetworkX libraries[4] in Python, where nodes are peers and edges are the neighbors of every peer. We used undirected graphs except for the computations of in-degrees, where we obviously built a directed graph with direction peer $\rightarrow$ neighbor. The switch to a graph structure allowed a series of operations, like the count of the connected components (or clusters), needed to interpret the shape of the network.

We approximated a couple of operation on the graph, in order to be able to execute it in a reasonable time. In particular the computation of the clustering coefficient was made using a function of NetworkX that make an approximation[LINK TO NETX CLUSTER]. The average path length was approximated computing the average path length of a number n of random couples of nodes.

4.2 GENERAL TESTS

In these tests we executed the simulation without interfering during the run, so no crashes or communication failures were induced. This simulations were done building a network of 20.000 peers with 20 neighbors and 10 of them exchanged at every cycle. In every simulation the Cyclon graph is compared to a graph with the same number of nodes and the same number of edges randomly distributed.

The first simulation (figure 1) compares the approximated average path length of the two graphs. We can see that after few cycles the avg becomes comparable with the one of the random graph. This is the optimum result in a network fully decentralized without a hierarchy.

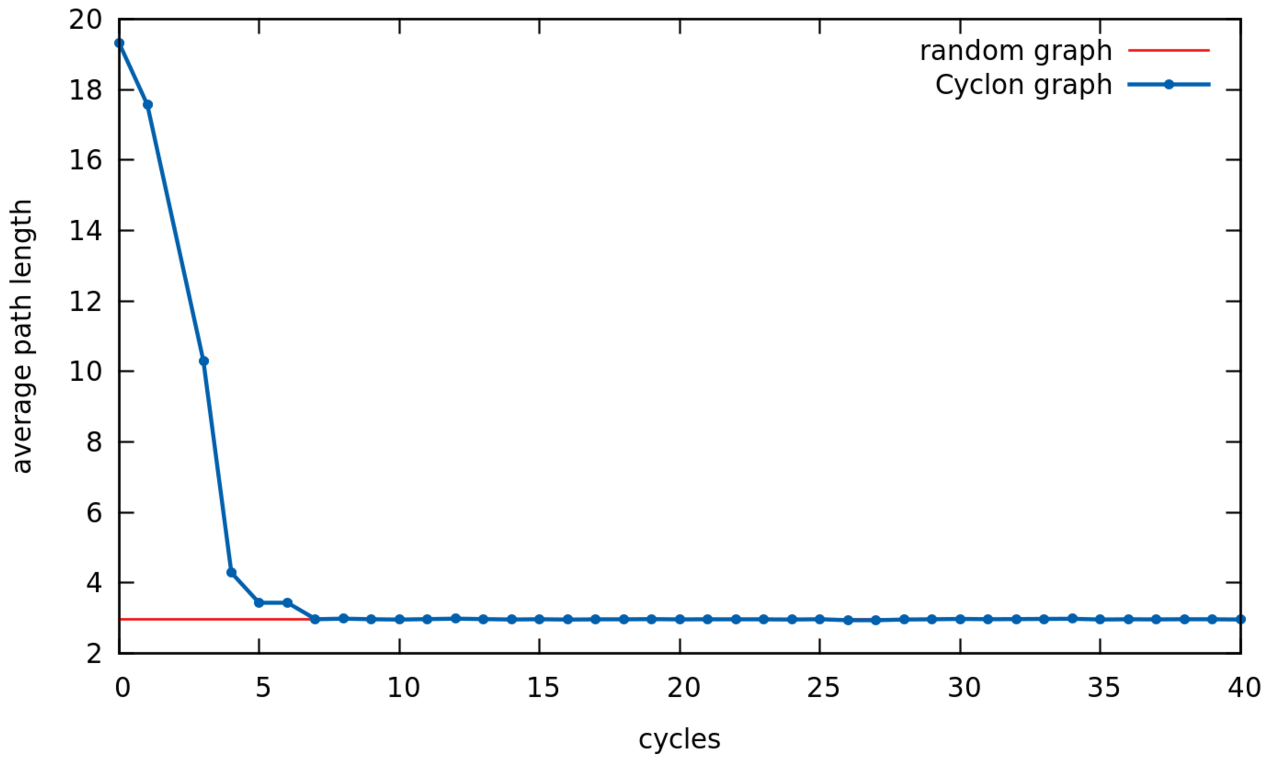

Figure 1: Average path length of random graph and Cyclon graph through cycles

The figure 2 shows the in-degree nodes distribution for the Cyclon and the random graph. The results show how the Cyclon graph has more nodes near the mean (20) than the random graph.

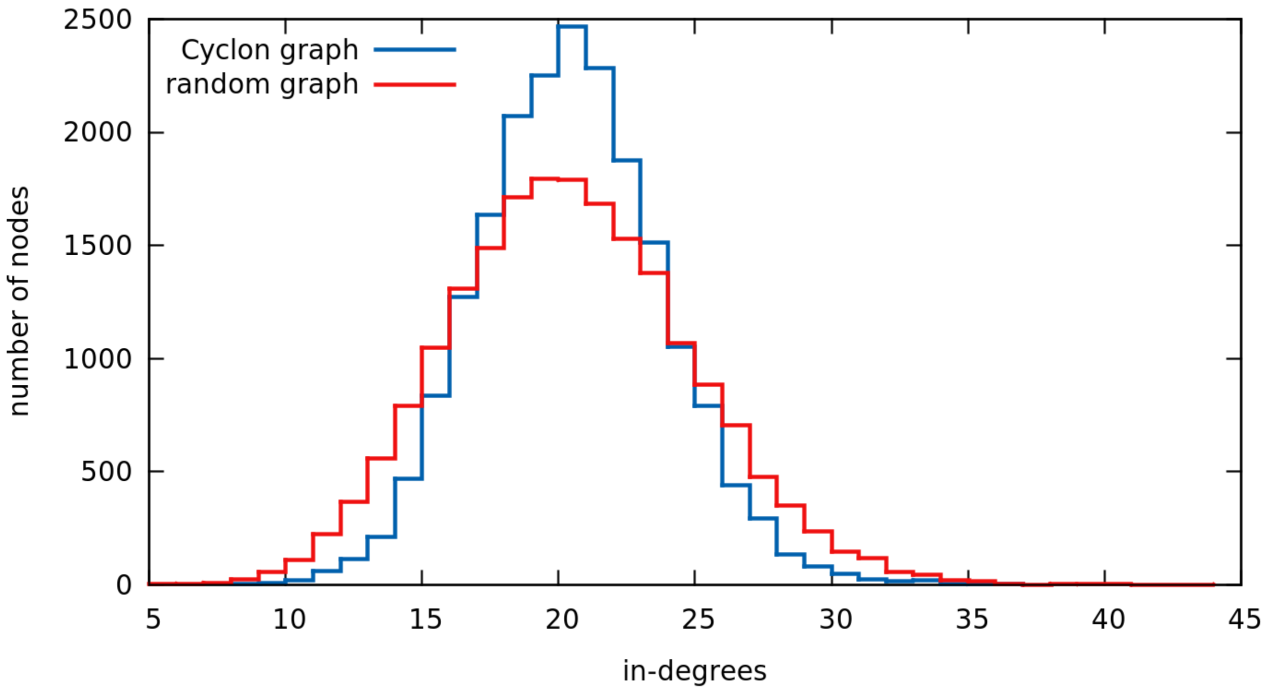

Figure 2: In-degree distribution for Cyclon and random graph

4.2.1 ROBUSTNESS TESTS

These simulation wants to show the robustness of the Cyclon protocol, that means its non hierarchical structure, that prevents damage in case of disconnection because there aren't fundamental peers for the network, and the resistance to massive node disconnection.

The first simulation (graph 3) regards the variation of the approximated clustering coefficient. The network, after few cycles, reaches the levels of the random graph. This is considered an index of robustness of the network, because of it means that there aren't nodes more important (significantly connected) than others.

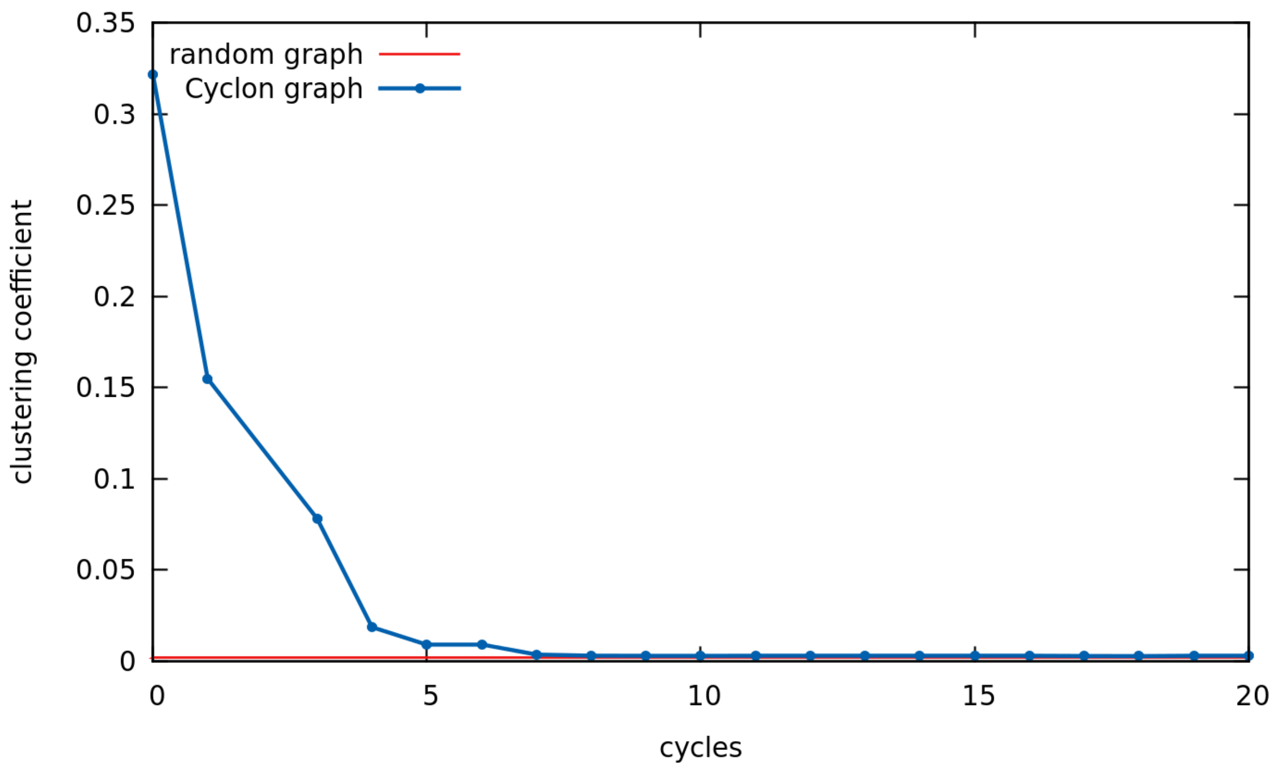

Figure 3: Clustering coefficient of random graph and Cyclon graph through cycles

Another robustness test is the count of the connected components of the graph after the removal of a certain percentage of nodes in the network. For every percentage tested we repeated for 10 times the removal of the nodes and made an average if the clusters generated. The graph 4 shows how the network remains connected until more than three quarters of nodes were removed, obtaining results comparable to the random graph.

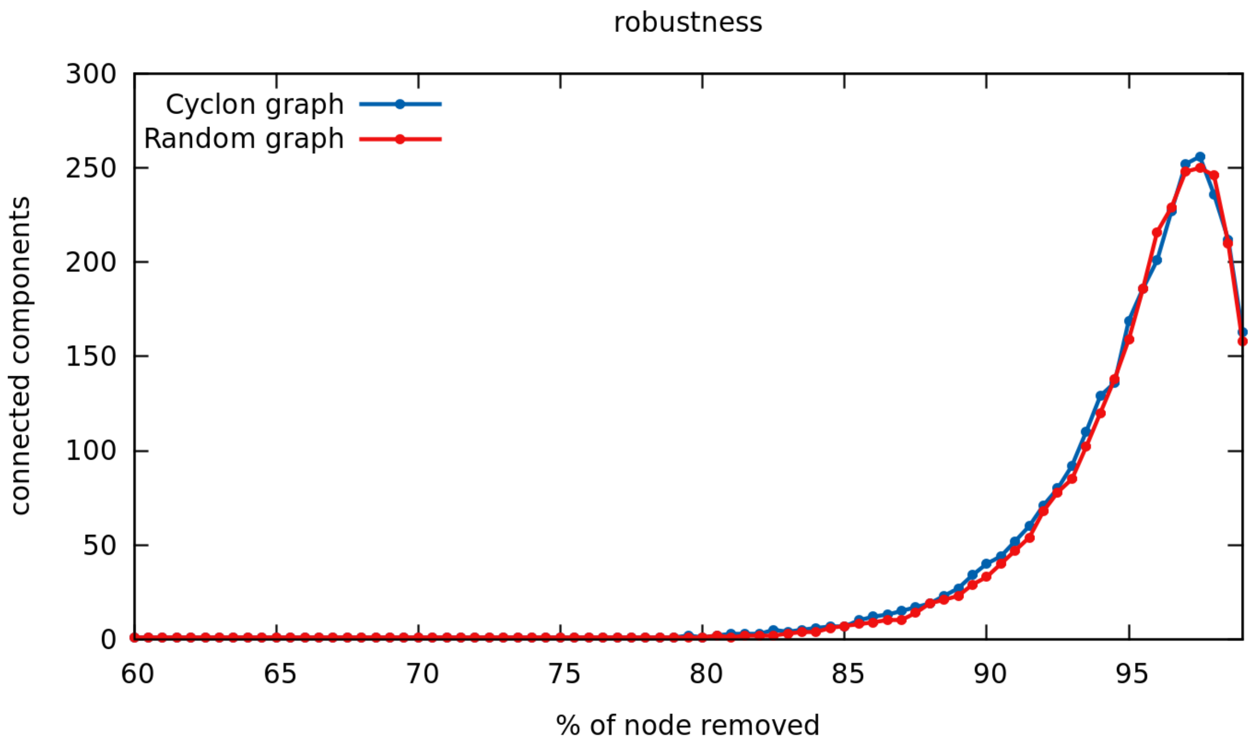

Figure 4: Connected components created by removing certain percentages of nodes in Cyclon nd random graph

4.3 LIVENESS TESTS

Another important feature requested to this type of protocols is the ability of maintain their network alive and free of garbage (references to peers still not present). We executed these tests with three different simulation varying the parameters of the neighbors kept and exchanged. The network had 5000 peers and 20-15-15 neighbor kept and 10-7-5 exchanged every cycle.

The first test was the ability of self-cleaning. The results (fig 5) show that the less number of neighbors allows the network to forget of the dead peers faster. The number of neighbor exchanged seems influence the speed of the process, in the sense that increasing the number of exchanged decrease the cycles needed for clean the network.

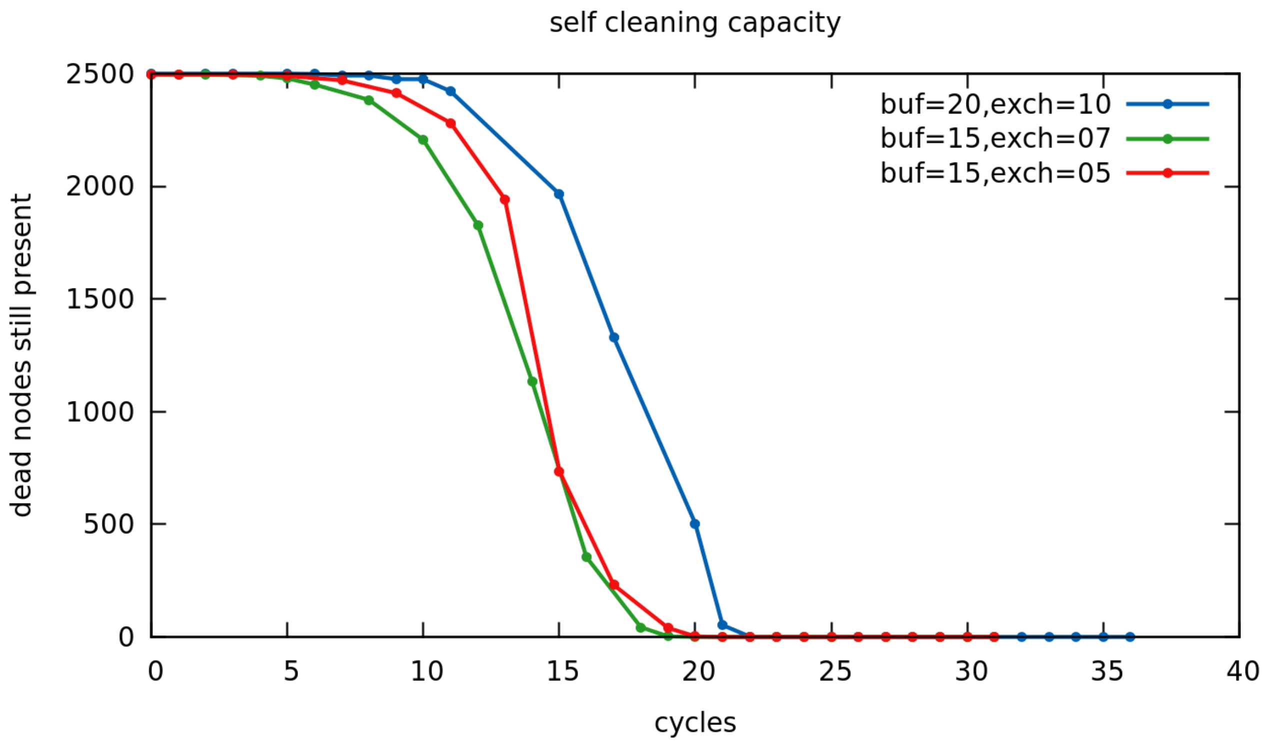

Figure 5: Number of cycles required to delete 2500 dead nodes

The last test executed was an hub attack test. We created 50 peers that exchanged only themselves, and measured at the end of every cycle the number of clusters after the removing of the malicious peers. As the figure 6 shows, the simple Cyclon protocol cannot resist and becomes polluted after few tens of cycles.

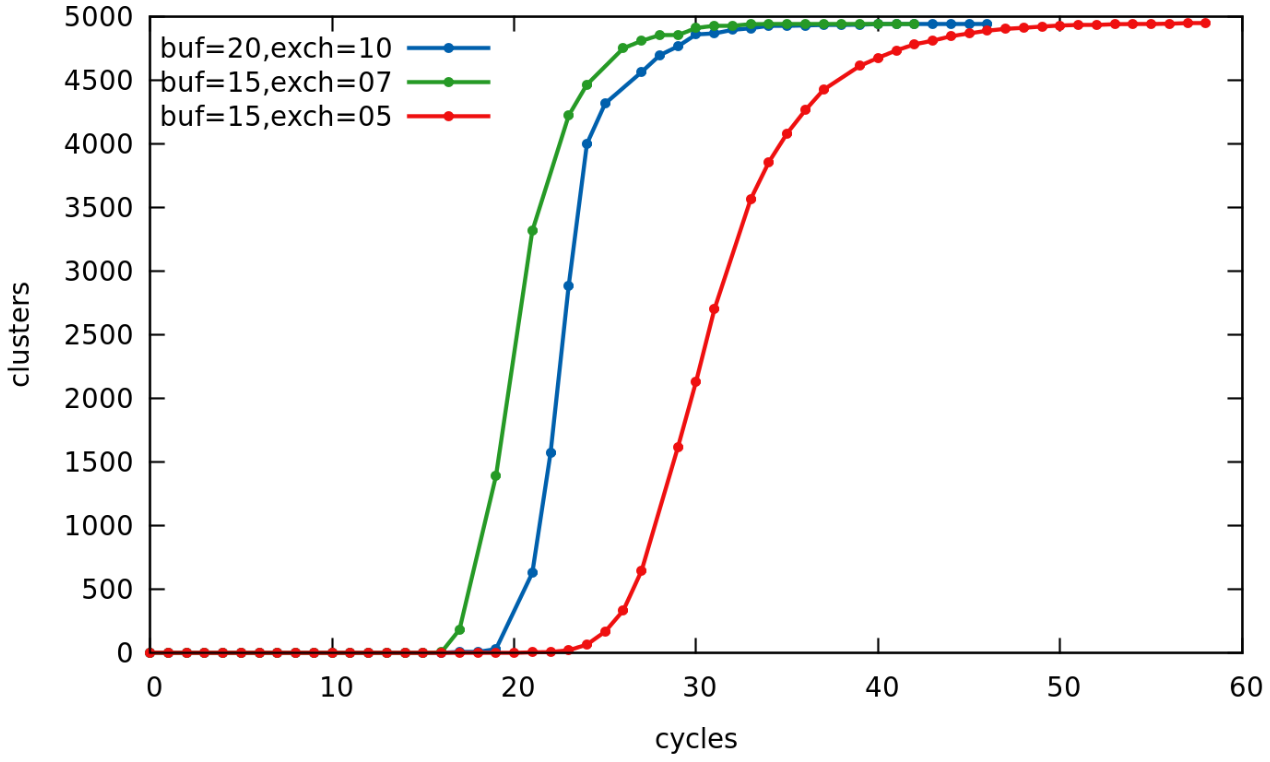

Figure 6: Number of clusters created at every cycle by 50 malicious peers

References

[1] The Akka framework, `http://akka.io/`

[2] Distributed Algorithms course, Montresor Alberto, slide "epidemics2"

[3] A Robust and Scalable Peer-to-Peer Gossiping Protocol by Spyros Voulgaris, M´ark Jelasity and Maarten van Steen,

[4] NetworkX, `https://networkx.github.io/`