Computer worm implementation in python3

1st Thomas Forsgren Ottestad
Institute of informatics
University of Tromsø
Tromsø, Norway
Tot005@uit.no

2nd Gustav Heide Iversen
Institute of informatics
University of Tromsø
Tromsø, Norway
Giv008@uit.no

***Abstract*—Having a better understanding of how computer worms work is an important step in keeping computer systems safe**

***Index Terms*—worm, distributed, informatics**

I. Introduction

This paper serves as an description of an implementation of computer worms. It will start with an explanation of what computer worms are, then how it was implemented. Then it includes a evaluation of how fast the computer worm grows and stabilizes after a certain number nodes are killed. The worm is implemented using python3.

II. Background

A. Computer Worm

[1] A computer worm is a piece of malware that can self replicate and spread to other computers. They most often spread themselves over a network utilizing weaknesses in the target system to bypass security. When a worm has infected a system it can use it as a host to find other connected systems to infect. This means that if the worm is left unchecked it will grow exponentially as long as there are systems available to be infected.

B. Election algorithm

[2] Election algorithms are a process of selecting a single process or node to do a set task. Election algorithms are often used to elect leaders or coordinators in distributed systems. Most election algorithms assume that each process has a unique ID or similar attribute that is used on the election process.

III. Worm

To clarify a running copy of the executable will be called a worm segment or segment, and the backdoor for spawning segments will be called worm gate or gates.

A. Structure

Every segment have a list of all the other segments and gates in the worm, these are used to ping and spawn new segments. This implementation is a centralized worm, where the responsibility of spawning and removing segments are controlled by one leader. The leader is decided by a id which it gets when initialized. Every segment have three threads, one for handling http requests, one to ping all the other segments and lastly one for the leader to control the size of the worm.

B. HTTP requests

The segment can take four different http requests, all of them are only used by other segments to get or send information, except Post set_max_segments which is only used by external clients.

1) GET segment_info: GET segment_info returns information from that segment including neighbours, max segments, amount of segments, own address, id and lastly if he is a leader. These are used when pinging to spread information. The most important part is the neighbour list so that the segments have the same neighbours in case of the leader crashing.

2) Get confirmed spawn: Get confirmed spawn is used when a non leader segment is spawned to notify the leader that he has spawned, and at the same time retrieving the neighbour and gate list.

3) Post kill: Post kill tells the retrieving node to shut itself down, stopping all threads. The killed segments does not notify anyone except the segment killing the node. This is only called by the leader when controlling the size.

4) Post set_max_segments: Post set_max_segments request is only handled by the leader, it sets the max amount of segments in the worm. This is what an outsider can send to the leader to control the size of the network.

C. Threads

1) Ping thread: The ping thread is where every segment pings each other to inform that they are alive and at the same time exchange information. It includes a for loop that loops over all of its neighbours and sends a GET segment_info requests, to check if its alive and to add neighbours that he does not have to his own neighbour list. If the neighbour does not respond then he gets removed from the neighbour list

2) Leader thread: The leader thread starts with a if statements to see if the segment is the leader, if it is it then checks if the amount of neighbour is greater or smaller than the defined max size of the worm. If it is smaller it sends a kill request to a random neighbour, if it is greater it spawns a new neighbour on a gate that does not have a segment. It checks how many segments the worm gate have by sending it a GET info request.

3) HTTP handler thread: This thread is so that every worm can handle concurrent requests without crashing or slowing down.

IV. Flow of the program

The first thing that needs to be done before spawning segments is to allocate the worm gates, and connect them as one network. It should not be necessary that every worm gate knows each other but all the tests that were ran in this paper have used fully connected worm gates. After allocating the gates one copy of the segment is sent to a gate using a http request, the segment is zipped before it is sent to the gate. The wanted size of the worm is passed through as argument in the http request. When the gate receives the segment it is spawned on that computer. The first segment is assigned as leader. The leader sets up all of its threads, and maps out all the gates by sending http get requests to its own gate, then all of the gates it receives. Ensuring it knows all gates in the network. After that he starts to see if there is room for any more segments, if so then it sends its own file as binary to one gate that does not have a segment. The id for the new segment is sent as argument to the gate, this is to ensure that there are no gates with the same id. When the new segment is spawned it sets up all of its threads, and send the leader a GET confirmed spawn request to the leader to notify the leader that he has spawned. The leader then puts the new segment in its own neighbour list. The leader then checks if there are room for any more segment, and goes through the same process. Until the max segment have been reached, then he waits for either noticing that a segment have been killed or if he gets a Post set_max_segments request to change the size of the worm.

V. Election algorithm

If a leader dies the other segment will find it out by not receiving a response when pinging him. Then the segments removes the old leader from its neighbour list and checks if his own id is smaller than all the other segment in the network, if this is true it sets him self as the leader. One thing that should be mentioned is that the segments does not know if who the leader is, they only know if they are the leader or not. This means that they have to check their id versus the neighbour every time a segment dies, but prevents sending request to notify the segment of a new leader.

VI. Performance testing

There were two different ways the worm segment was tested, one that timed how long it took to grow the worm from 1 to N segments, another that timed how long it took for the worm to recover after killing 1 to N worm gates. All of the tests are ran by first setting up the worm gates, and then sending http request in python using the requests library to the gate and leader.

A. Test result

1) Time to grow worm size: To be able to test how long it took to grow a worm of size one were set up on. Then a http set_max_segments request was sent to the leader telling him what size to grow to. The leader does not respond before it has as many neighbour plus him self as the wanted size. The timer starts right before and ends right after the request, the results are then dumped to a json file. Result is displayed in figure 1.

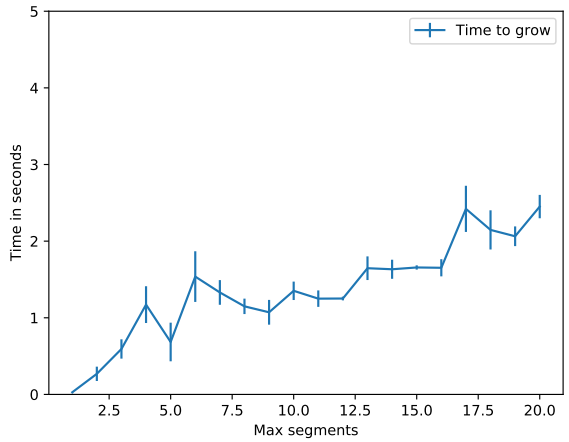

Fig. 1. Time to grow worm from 1 to max segments

2) Recover from killing segments: To test how long it took for the worm to recover a worm of size 10 were set up. Then a request to kill N gates were sent to the gates. To see if the network had recovered a request to all the gates in the worm to see if they have one segment each. The timer were started from the first kill_worms to when the all gates have responded that they have one segment, the results are then dumped to a json file. Result is displayed in figure 2.

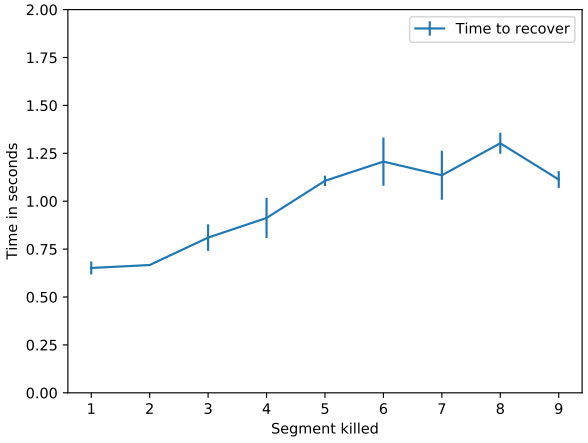

Fig. 2. Time to recover from killing segments

B. Evaluation

As can be seen in figure 1, the time it takes increasing the number of worms from 1 to n does is very consistent. It increases slowly in a linear fashion. This is expected since all of the growing responsibility is done through the leader, meaning there are should not be any difference from growing

from size 2 to 3 than 6 to 7. The time to recover the network after killing gates also grows in a linear fashion. But one thing that is surprising is that the worm recovers a bit faster than the worm grows. The difference is not that large so it might be that the nodes that were allocated for recovery were faster than the ones for growing. The reason they didn't use the same, is because they used different amount of worm gates.

VII. Future work

There are features that could be added to make the worm faster and make it so that it does not as many request as it uses. Instead of every segment pinging every segment the leader could have been the only one pinging, and the others wait for a ping from the leader and if they does not receive a ping within a time they could start a form of election. This would reduce the amount of request between the segments.

There could have been performed tests that checks how the worm handles that the leader crashes, this is not tested in 2.

VIII. Lessons learned

The main lessened learned implementing worms using python3, is how important it is to use correct exception handling. Making sure that the segments doesn't crash when sending request to segments that might have been killed, but at the same time not hiding error that should be fixed.

References

[1] Michael T. Goodrich and Roberto Tamassia. *Introduction to computer security*. Pearson, 2018.

[2] *Election Algorithm*. URL: https://www.sciencedirect.com/topics/computer-science/election-algorithm.