The Worm

Ka Hin Lau and Narae Park
Department of Computer Science, UiT The Arctic University of Norway

***Abstract*—Fault tolerance is important in distributed systems to keep the systems always alive when failures occur such as machine or network shutdown unexpectedly. In this assignment, we are going to create the worm as a distribute program, and keep the system alive in the face of failures.**

***Index Terms*—computer worm, fault tolerance, distributed program**

I. Introduction

THE worm is a program that replicates itself to other machines over a network, created in the development of distributed computing. Introduced in 1982 by John F. Shoch and Jon A. Hupp at Xerox Palo Alto Research Center[1], the worm was originally designed to find computers that were idle over the network, allocate tasks, and share computations to increase the efficiency of the entire network. However, Worms gained worldwide attention in 1988 when the worm developed by Robert T. Morris, a graduate student at Cornell University, rapidly replicated the Worms to numerous computers connected to the Internet, causing great damage. Worms basically consume computer resources and affect network traffic, and they also cause malicious damage to computers, such as deleting or encrypting files, and leaking data, depending on the payload designed that performs operations other than spreading. As a result, the spread of the worm has served as a wake-up call for computer security[2]. In this assignment, through the implementation of the worm, we will explore how the worm can replicate itself to other machines through the network.

II. Description of system

A. Architecture

We made the structure of the worm by similar to the ring-shaped chord system[3]. Each worm segment has a predefined position inside the ring, and each of them maintains a state, which is a list of all other worm segments' host-name and port.

In the initialization phrase, each worm segment spawns the next segment in the ring through the worm gate, then it will send an update about its worm position, host-name and port to all other worm segments it is aware of, such that all worm segments can have latest information about that worm segment at that position. The target is reached when last worm segment in the ring is spawned. In addition, all the worm segments same information about other worm segment after all the worm segments receive the update from the last worm segment.

Each worm segment is responsible for checking the health of the next worm segment periodically. Only when the next worm segment is down, the worm segment will spawn a new worm segment as its next worm segment.

This approach can distribute tasks among the worm segments compared to a single leader in charge of all spawning and communication tasks. However, if some consecutive worm segments are killed at the same time, it may take more time to recover back to the target size because the worm segment is recovered one by one.

Communication between worm segments occurs when a newly spawned worm segment send an update about its index and name(address) to other worm segments that it is aware of. And the communication also happens when each segment periodically checks its next worm segment. Therefore, it is possible to maintain a worm of the entire target size even with relatively little communication between the worm segments. This is based on prior coordination that each worm segment is only responsible for the next worm segment in the entire worm.

There are two RESTful HTTP API in the worm segment code for their communications.

The first one is a POST with path "/done_spreading", which is used to send an update to another worm segment.

The second one is a GET with path "/info", which is used to get the information about the worm.

B. Design

The process of generating and maintaining a worm with segments of the target size was designed as follows.

1) Generating a worm with segments of the target size
    a) Open one or more available worm gates.
    b) Create one worm segment through the worm gate (target size and index position are provided as arguments).
    c) The worm segment creates an empty list of target size and fills in its name(address) in its index position.
    d) If there are other worm segments in the list, the newly spawned segment send its index and name to other segments, such that all the worm segments can update its own worm segments list with the new worm segments in the corresponding position.
    e) If the next index in the list is empty, the worm segment spawns the next worm segment through a worm entrance using a neighbor worm gate with randomly selected port.

    When all indexes in the list are filled, it means that a worm with segments of the target size has been generated.

2) Maintaining a worm with segments of the target size
   a) Each worm segment periodically sends a request to the segment of its next index to ensure that the next worm segment is alive.
   b) If there is no response (which means the worm segment is dead), repeat the process of 5 to re-spawn the next worm segment.

   Through a periodic stabilization process, the worm with segments of the target size is maintained.

It is not required that the worm gates in the network know each other. When a worm segment spawns a new segment, it uses its worm gate's neighbor worm gate (if there is no neighbor, it uses its own worm gate). Also, therefore, one worm gate can have multiple worm segments.

C. Implementation

1) Overview: The code of the worm segment was made using Python as the programming language. The worm ran on The University of Tromsø's cluster of nodes. The hardware specifications of each node can vary, and the distance between nodes also vary depending on which node communicates with which node. In the consideration of this variance, the experiment was performed several times and the results including the error bar are provided in section 4.

The first worm segment is created through its worm gate by a "POST /worm_entrance" request to that worm gate. In the POST request, there are five arguments and the worm segment code as the data. The first argument is the port of the worm gate, which allows the worm segment to duplicate itself to another machines through its worm gate. The second argument is target size, which control the size of the worm segments. The third argument is the port number the worm, which is used to communicate with other worm segments. The fourth argument is the worm segment position named "worm_position". For example, the first worm segment has the position of 0, the second worm segment has the position of 1 and so on. This position sticks to that worm segment forever. The default value is 0. The fifth argument is a string named "worm_all", which contains all the previous worms segments that the worm segment knows so far, and the each worm segment is separated by a comma. For instances, the first worm segment has an empty string because there is no previous worm segment before it. The "worm_all" of the second worm segment is "worm0", and "worm0,worm1" for the third worm segment and so on.

Once the five arguments and the worm segment code are delivered to worm gate, the worm gate will spawn a sub-process, and then start executing the code and pass all the arguments into the code. If the target size is 0, then the code will just end, otherwise the code will start running.

The worm segment code will first run the function "start_spread" to prepare the five arguments for the next "POST /worm_entrance" request to another worm gate and maintain its own worm segments list, named "worms" as state. After that, it will start a HTTP server to communicate with other worm segments.

2) Inside the function "start_spread": First, it makes a "GET /info" request to its worm gate to get the neighbors worm gates. Then it will pick a random neighbor worm gate as the target worm gate and generate a random port as the target worm segment port number. Next, its own worm segments list is initialized with the size of the target size and empty string for each element inside the list. If it is the fist worm segment and the "worm_all" is empty, we will update both the "worm_all" and the element of "worms" list with its worm position to its name. For instances, if the target size is 3, then the "worm_all" becomes "worm0" and the "worms" becomes ["worm0", "", ""]. Otherwise, if the length of "worm_all" is smaller than the length of "worms" list (in other words, the worm segment has information about the previous worm segments and the worm segment is still in the initialization phrase), its name is added to "worm_all", and the "worms" list is updated with both its name and the name of the previous worm segments. For example, in the case of the second worm segment with target size 3, the "worm_all" becomes "worm0,worm1" and the "worms" list becomes ["worm0", "worm1", ""]. If the length of "worm_all" is larger or equal to than the length of "worms" list (in other words, the worm segment is not in its initialization phrase), the old worm segment is replaced by the new segment in both "worm_all" and the "worms" list. For example, if the worm segment is named worm "1b" with worm position 1, then the "worm_all" changes from "worm0,worm1,worm2" to ""worm0,worm1b,worm2" and the "worms" list changes from ["worm0", "worm1", "worm2"] to ["worm0", "worm1b", "worm2"].

After that, we prepare the worm position for the next worm segment. Since it is a ring topology, we just add the worm position by 1 for the next worm segment, and set it to 0 if the current worm segment is the last one by checking the target size. Next, if next element in the "worms" list is an empty string, meaning it is still in the initialization phrase, then it will send "POST /worm_entrance" request to the neighbor worm gate selected earlier together the updated five arguments and code itself as data. Otherwise, if next element in the "worms" list is an empty string, the target size is reached and the worm segment will stop spawn another worm segment.

Finally, the worm segment will send a POST "/done_spreading" to each worm segment inside the "worms" list expect itself and element with empty string. The target worm segment will receive the name and the position of the source worm segment such that it can update its own copy of "worms" list by the position with the new name.

3) Synchronization and Recovery: In initialization phrase of growing the worm segments, each worm segment is responsible for spawning the next worm segment once until the target size is reached. Also, it sends an update to other known worm segments after send the POST request to the spawn the next worm. As a result, after all the worm segments receive the update from the last worm segments, all the "worms' list of each worm segment should be the same such that synchronization is achieved.

However, if any worm segments are killed after the initialization phrase, the target size is not fulfilled and the "worms"

list of each worm segment becomes out-of-date. To solve this, we implement a stabilization function inside the worm segment code and run it with time interval T with default value 1 second.

In the stabilization function, each worm segment is responsible for checking the health of the next worm segment by sending a ”GET /info” request to its next worm segment for every T seconds. Only if the next worm segment is down, then it will simply send a ”POST /worm_entrance” request to another worm gate to spawn a new worm segment. The new worm segment will then send an update to all other worm segments such that other worm segments will update their own ”worms” list by replacing the old worm segment with the new worm segment according to new worm position.

III. Evaluation

We conducted two experiments to measure the performance of our worm program. To run the experiments, separate shell scripts from the worm program were used.

In the first experiment, the time from the generation of one worm segment to the completion of a worm having segments of the target size ”n” was measured. The timer was started after sending a command to create the first worm segment. To determine the completion of the worm, the number of segments of each worm gate was checked and summed within the loop, and if the number was equal to or greater than the target number of sizes, the loop was terminated and the timer was stopped. The time was calculated in milliseconds using *$(date +%s%N)*. In order to see the effect of other variables such as the machine node’s capability, the time was calculated when only two fixed machines were used and the machines of the target size were used as warm gates. The target size of the segment was set from 2 to 20, and the mean and standard deviation were calculated by performing five experiments for each.

In the second experiment, the time from when one or more segments of a worm with segments of target size were killed to recovering to a worm with segments of the original target size was measured. The timer was started after sending a command to kill to the worm gates of the number to be killed. To determine that the worm has recovered with segments of the target size, as in the above experiment, the number of segments of each worm gate was checked and summed within the loop, and if the number was equal to or greater than the target number of sizes, the loop was terminated and the timer was stopped. In order to minimize the influence of other variables such as the machine node’s capability, 10 fixed machines were used for the worm gates. The target size was set to 10, and the number of segments to be killed was set from 1 to 9. Five experiments were performed for each to calculate the mean and standard deviation.

IV. Results

The results of both experiments are shown in the following plots. The x-axis represents the target size of segments, and the y-axis represents the elapsed time (millisecond). The average of the elapsed time at each size is shown as a line graph, and the standard deviation of the elapsed time is shown as an error bar.

A. Generating a worm with segments of the target size

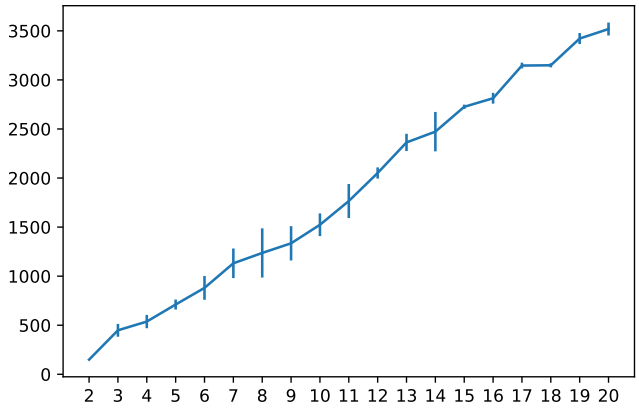

Fig. 1. Time to grow from 1 to *n* segments (with 2 worm gates), *n=2 20*

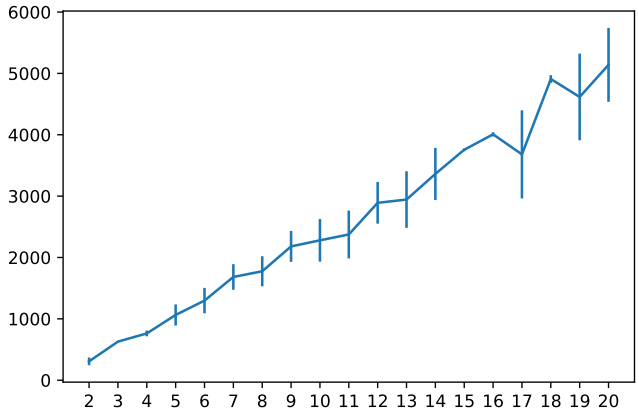

Fig. 2. Time to grow from 1 to *n* segments (with *n* worm gates), *n=2 20*

Figures 1 and 2 show the elapsed time to generate a worm with segments of target size from one segment. Figure 1 is the result when the worm gates are set to two fixed machines (maintaining the same port) to minimize the impact of the machine. Figure 2 is the result of experiments with increasing the number of worm gates according to the number of target sizes so that each worm gate has one worm segment. When increasing the number of worm gates, previous machines were kept intact and a new worm gate was added to minimize the impact of machine changes.

B. Recovering a worm with segments of the target size

Figure 3 shows the time it takes to recover to a worm with segments of the target size when n segments are dead. In order to minimize the influence of the machine, the machines used in the experiment were kept the same.

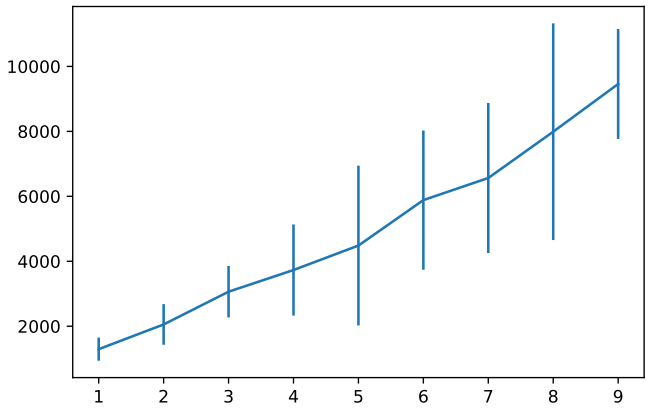

Fig. 3. Time for worm of size n to recover from k segments killed, *n=10, k=1 9*

V. Discussion

From the figures shown above, the time needed for both the worm to grow and to recover increases almost linearly when the target size increases. This behavior is expected because more the worm segments, more the communications are needed between themselves.

The worm segments are connected in a ring topology, with each worm segment having its own position inside the ring. Each worm segment spawns the next worm segment until the target size is reached. This is similar to the Chord protocol. However, in Chord, when one node fails, its neighbors will just discard the failed node and then updates its neighbors to the closest neighbors to maintain the ring, so the size of the ring will gradually decrease. In our implementations on the worm, if a worm segment is down, in order to maintain the target size, the previous worm segment of the killed worm segment will spawn a new worm segment as its next neighbor. Then the new worm segment will update all other worm segments it knows about itself and its position.

In the case of killing worm segments which are positioned next to each other, the recovery performance is degraded because only the previous worm segment is aware of it and the new worm segments are generated one by one. For example, if the worm segments with positions 2, 3, 4, 5 are killed, then only the worm segment with position 1 is aware that the worm segment 2 is killed, so worm segment 1 will spawn a new worm segment in the position 2. Then after a time interval the worm segment 2 is aware of the failure of worm segment 3, so it will spawn another worm segment in position 3. So the recovery process is only considered done until a new worm segment 5 is spawned. Therefore, looking at the recovery time in Figure 3, it can be seen that the deviation is quite large. This is presumed to be because the recovery time is affected by which nodes in the network topology are killed.

Comparing Figure 1 using two fixed machines and 2 using a machine with the number of segments in the worm growing experiment, both show linearly increasing graphs, but in Figure 2, the time increases by about 1.5 times, and the fluctuation of the graph is slightly more. It can be said that the communication time with other machines is reflected. In addition, it can be said that there is a difference in worm execution time depending on the machine.

VI. Conclusion

We implemented a worm as a distributed program that can self-extend to have segments of a target size and recover to have segments of a target size again even when some segments are failed. We then measured the time it takes to extend the worm to have segments of target size and the time it takes to recover back to its original size in case of failure of some nodes.

References

[1] Shoch, J. F., Hupp, J. A. (1982). The “worm” programs—early experience with a distributed computation. Communications of the ACM, 25(3), 172-180.

[2] Orman, H. (2003). The Morris worm: A fifteen-year perspective. IEEE Security Privacy, 1(5), 35-43.

[3] Stoica, I., Morris, R., Liben-Nowell, D., Karger, D. R., Kaashoek, M. F., Dabek, F., Balakrishnan, H. (2003). Chord: a scalable peer-to-peer lookup protocol for internet applications. IEEE/ACM Transactions on networking, 11(1), 17-32.