INF-3203
Advanced Distributed Systems

Assignment 2

Andreas Brunes, Magnus Kanck

02.05.2022

1 Introduction

In this report the design, implementation and evaluation of an "existential worm" will be presented. The worm will access a distributed system via a known vulnerability and then grow to a target length. The performance of the worm will be evaluated by measuring the time taken to grow to a specific size, and the time taken to regrow as the connection to segments are lost.

2 Design & Implementation

All worm segments, regardless of its role in the system, will use the same class *Jorm*, shorthand for the mythical sea serpent "Jormungandr". To keep with the nautical theme, the communication approach, described in Section 2.2, will be called "flooding". The *Jorm* class will contain all necessary methods to allow given segment to take on the role as a *leader* or *follower* at any time.

A single worm segment will act as the *leader*, responsible for coordinating with, and spawning, the other segments. At regular intervals the leader will message the non-leader segments with information. The message will act as a ping to notify the segments that the leader is still active, and to disseminate updated information about the state of the worm, in case the leader goes down. *Follower*, or non-leader, segments will generally have two tasks; to inform the *leader* segment that they are still alive, and to receive updated information from the *leader*. In the special case where the *leader* is unresponsive for long time, the *followers* will have the additional task of electing a new leader as well.

2.1 Leader election

The election algorithm will be a simple procedure where each *follower* picks the first entry in their local list of active segments as the new leader. It is the responsibility of the leader to make sure that each *followers* active segment list is synchronized with itself.

2.2 Messaging

All messaging will be non-blocking and asynchronous, which will be achieved using a UDP protocol. A segment will not care if a given message is received or not, it will simply "fire and forget". This is to ensure that network traffic remains low and that segments do not spend a lot of time attempting to handshake with unresponsive ones. The leader will communicate with the rest of the worm by broadcasting information to all segments at roughly the same time, then wait for a bit to allow the *followers* to send messages in return. The *leader* does not need to hear from a segment to send it a new message, the same applies to the segments. This procedure can be thought of as an intermittent message "flood" where information is spread out from the *leader* to the *followers*. Then, an intermittent localized "flood" is sent from each segment to the leader.

The wormgates from the precode will only act as as entry points for injecting segments into the distributed system. After the segments have spawned they will open their own UDP ports for communication. Naturally, these ports do not expose the system to the same vulnerability as the wormgates, but only serve as communication channels.

2.2.1 Measuring responsiveness

The messaging system eschews handshaking to reduce traffic, therefore the *leader* must have a send/receive counter for each segment it controls. The *leader* increments this value each time it sends a message to a *follower*, and resets it if it gets a message from that same segment. If the counter reaches a certain threshold the leader will infer that the given *follower* has crashed. It does this by checking if the send/receive counter of the segment with the highest value is larger than the unresponsiveness threshold. If yes, the *leader* moves its entry from the *active* table to the *bucket*.

Followers will only give the leader one chance to communicate, and if the timeout occurs the *follower* infers that the leader is dead and will elect a new *leader*.

2.3 The *Jorm* class

Each segment will have access to three localized tables, *active*, *bucket* and *available*. The entries of these tables will hold mappings between wormgate ports and segment communication ports. As only one segment can spawn for each wormgate, these mappings will be one-to-one. Entries in the *active* table are worms that have spawned and are still responsive, the *bucket* table contains entries of segments that the leader has inferred have crashed and the final table, *available*, are wormgates that have yet to be used. Depending on what role a given segment has, it will enter one of two logic loops; the "leader_logic" or the "segment_logic".

Algorithm 1 Leader Logic loop psuedocode

```
 1: while True do
 2:     if $length < target$ then
 3:         Choose available wormgate
 4:         Update active followers
 5:         Spawn segment
 6:     end if
 7:     Update active followers
 8:     Listen to followers
 9:     if $\max(active_{send/recv}) >$ threshold then
10:         Move unresponsive follower to bucket
11:     end if
12: end while
```

Algorithm 2 Follower Logic loop psuedocode

```
1: while True do
2:     Listen to leader
3:     if timeout then
4:         Start election
5:     else
6:         Update worm state
7:     end if
8:     Ping leader
9: end while
```

2.3.1 Spawning new segments

The *leader* will continually check if the worm has reached its target length and will try to spawn new segments until the target is reached. Just before spawning a new segment the *leader* will update the already active segments with the future status of the worm. This is done in case the *leader* crashes after spawning a new segment and before the regular update "flood". Without the pre-spawn update, the active segments would have no way of knowing about the new segment. This pre-spawn update, however, raises another issue that is resolved by the send/receive ratio. The newly elected leader will notice that the wormgate where supposedly a new segment has spawned is unresponsive, and throw it in the *bucket*. If the *leader* later on runs out of available gates, it will attempt to use the entries in the *bucket* instead.

3 Experiments

The time taken for a worm to grow to its target size will be measured by the leader. The time starts from the time the *leader* segment first enters the *leader* logic loop "leader_flood", and ends when the *leader* has gotten a message from each unique segment it has spawned.

The recovery time is measured in the same way. The time begins when the *leader* infers that one of more of its *followers* have crashed. Again, the timer ends when it has received a message from all the new segments it spawned. Random worm segments are killed in bursts via a python script "random_kill.py" that excludes the *leader* from the list of segments that can be killed. This is done to simplify the taking of measurements.

All tests presented in this section have been run on a computer with the specifications listed in Table 1.

Specifications
CPU: Intel(R) Core i5-8265U @ 1.60GHz RAM: 8GB

Table 1: Computer specifications which are used to run tests.

3.1 Average grow time

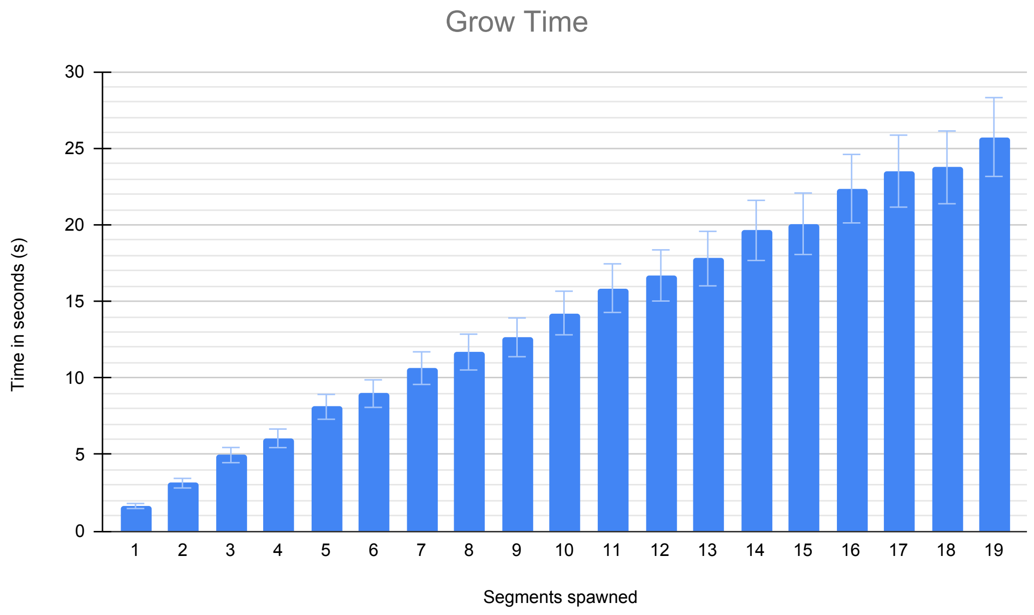

Figure 1: Plot of average time grow worm of size n, where n = 2 . . . 20.

Figure 1 shows the average grow time of a worm with variable target size. For each data point the experiment was run three times, and the environment was reset each time. In addition, segment crashing was disabled to eliminate noise from the measurements.

3.2 Average recovery time

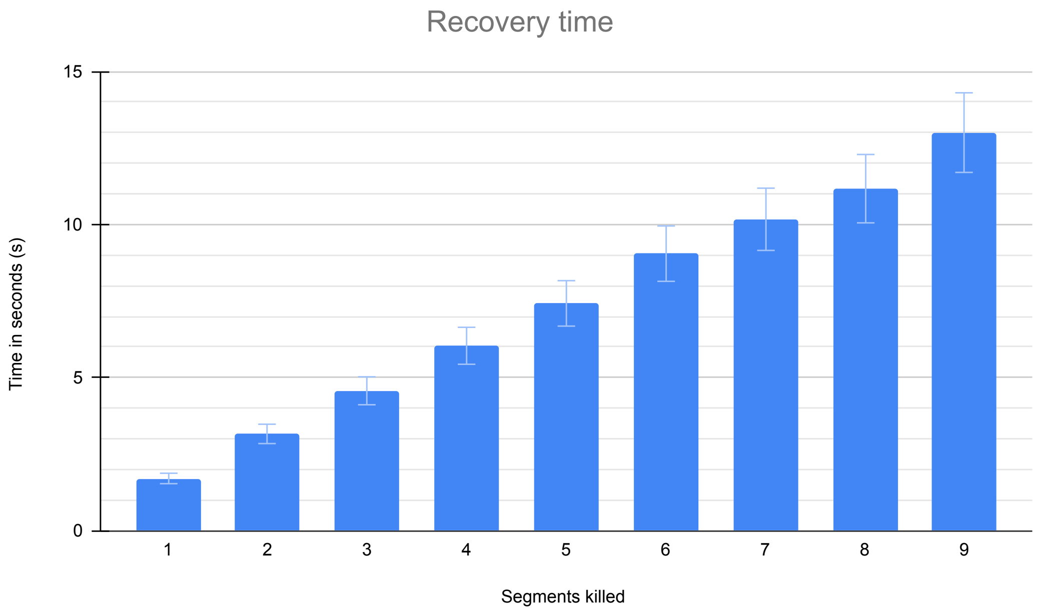

Figure 2: Plot of average time for worm of size 10 to recover from k segments killed, where k = 1 . . . 9.

Figure 2 shows the average time for a worm of size 10 to recover from a variable number of segments crashing. The average was taken over three tries, the environment was reset each time and the *leader* segment was left active throughout each run. The nodes to be crashed was chosen uniformly at random.

4 Discussion

From the results in section 3 it seems the average grow time increases linearly as the target size increases. This might be due to the way new segments are spawned, via a python ”os.system” call. This command starts a sub-process, which could add a considerable overhead. Additionally, the *leader* will only spawn one new segment at a time. In other words, it will coordinate with the other active *followers* in between each spawning sequence.

The average recovery time (Figure 2) is also linearly increasing with the number of segments killed. This might be because the *leader* will only register one segment as crashed at a time. Therefore, it must go over the whole *leader* logic loop before registering a new one as crashed.

4.1 A leader crashing after spawning

Due to the way communication between segments is handled ("fire and forget") an edge case might arise if the *leader* crashes after spawning a segment. If one or more of the active *followers* fail to receive the update before a segment is spawned and the *leader* crashes after spawning, then those *followers* will not know about the new segment. However, since *leaders* are elected based on time of spawning (first to last) all *followers*, even the newest, will elect the same *leader*. The newest segment will establish contact with the new *leader* regardless if the *leader* knew of it or not, which will then add it to the list of active *followers*.

4.2 A splitting worm

If, after a leader election, one or more *followers* for some reason did not receive any message from the new leader before the timeout expired, they would start a new election and choose a new leader. In this case, the worm would split up into two separate systems. Each leader would infer that the segments in the other system have crashed, as they receive no communication from them. This would lead to a race between the two leaders to "capture" the remaining available wormgates in order to reach the target size, which would quickly exhaust the list of available gates.

4.3 Packet loss and Responsiveness

During testing, the send/receive counter used by the *leader* showed that packet loss increases with the number of active segments. This is most likely a result of how the communication approach only accepts one message for each iteration of the *flooding*. More segments will therefore lead to packets competing for the slot in the receive buffer and the send/receive ratio will skew.

Furthermore, the send-receive counter, as previously mentioned in Section 2.2.1, is used in conjunction with a threshold to infer whether a segment has crashed. This threshold needs to be set by the *leader* to accommodate the size of the worm because of the above mentioned communication approach. In the current implementation, the threshold is set to 50 through trial and error for a worm size of 20.

5 Future work

To reduce the potential overhead when growing worms, the "os.system" call could be changed out for a http request instead by using the "urllib" python module, which should be faster as it would not start a sub-process. Alternatively, the *leader* could create a new thread to handle spawning of new segments, giving it more time to coordinate with the active *followers*. Lastly the "subprocess" python module could be used instead, which provides a more lightweight method that retains the same behaviour as "os.system".

The bucket table has been implemented to contain segments that the *leader* have inferred to be crashed segments. However, the logic for retrying these wormgates when there are no more available gates has been omitted due to time constraints. The logic includes trying to re-establish a connection while below the target size and spawning new worms.

The actual implementation of the leader "flood" communication approach deviates somewhat from the proposed design. Instead of broadcasting updates to all *followers* concurrently, the *leader* updates them sequentially instead. This was done due to time constraints, and an untimely case of covid in the project team. Future work on the leader "flood" could be to make it concurrent by way of threading.

6 Summary

In this assignment the design, implementation and evaluation of an existential worm was presented. The time to grow increases linearly as the target length increases. The time to regrow also increases linearly with the number of segments that were killed. Due to time constraints some parts of the proposed design were not implemented, such as retrying unresponsive wormgates and concurrent communication with *followers*.