UNIVERSITY OF TEHRAN

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Distributed Systems : Computer Assignment 1

Author:
Seyed Mahdi Hosseini (ID: 810197652)

Date: December 19, 2021

Contents

1 Introduction

All codes are available on GitHub
In this project, we have to provide a multiprocess program to run and evaluate distributed algorithms discussed in the class. We started from a starter code which is available on GitHub and was given by the TA of the course.
We are supposed to make some changes in the starter code to:

- 1: run echo algorithm with extinction for anonymous networks
- 2: evaluate time and message complexity for different types of networks.
- 3: Implement a selected paper method and compare the results.
- 4: Implement part 2 by using Shadow Discrete-event Network Simulator.

The starter code uses RabbitMQ for process communication. Therefore, we need to install it with this guideline first. The starter code consists of four main python files. The starter code consists of four main python files.

- **runner.py** : this program manages the network and processes. We can execute this program by passing needed arguments to it, such as ”input.in” which is an input file that contains the number of nodes and edges of the network graph. During the execution, the network graph is constructed using networkx. After constructing the graph, this program forks and makes N number of child processes (as nodes of the network) and waits for them, which means that it waits for its child process to terminate.
- **node.py** : this program, which is forked from the runner process, simulates a node in the network, and we have to make some changes in each section to run algorithms. in this program, every node uses a world which is the insight of that node from the network and contains network-related status. Because in this project, we just have to run the election algorithms, this program starts the algorithm in each section and tries to listen for upcoming messages.
- **world.py** : As said before, every node uses a world that is the insight of that node from the network and contains network-related status. We have two types of world subclass of abstract class AbstractWorld: 1-SimulatorFullView: this class has information of all nodes. 2-SimulatorOnlyNeighbors: this class has just neighbor's information and does not know about anything about others.
- **algorithm.py**: The main activities of this project have been done over this program. The approach is that when every node receives a message, it calls a function to process that message. This function proceeds the message based on the algorithm implemented in this file.

2 Echo algorithm with Extinction

In this part, we are supposed to implement the Echo algorithm with Extinction for an anonymous network [2]. In all sections, I will explain the changes I added to the code and their function for each file.

2.1 node.py

According to the algorithm, all nodes are active at first and want to be the leader. Then in this file, every node starts a new round (round 0). After starting round 0, each node listens to receive a message and process the message. Starting round is implemented in world class

2.2 world.py

Although We have to use SimulatorOnlyNeighbors class in this section because every node knows its neighbors in the anonymous network, I use some methods from the super class(SimulatorFullView), such as constructor. At first, because all nodes know the size of the network, I added a new attribute to this class to store the number of all nodes in the network(number_of_nodes_world_map). Another attributes that were added to the world class are current_leader_id and current_leader_round, which keep the last id and its round that the node has seen and accepted based on the algorithm. The new received message proceeds and can change these attributes based on the algorithm.
To start a new round, I wrote start_round method in SimulatorOnlyNeighbors class. In this method, which is called by every node at the beginning of the program for starting the first round, the world selects a random id between 1 and n (the number of nodes in the network). After selecting, the world sets the current round of this node to the argument that was passed to the method, then the world constructs a new wave message that includes the round of this node and the new id and then sends it all neighbors
When a node receives a new message, it processes the message in the algorithm file. In this section, I implement a more logical algorithm for terminating processes: The leader node starts a new wave for termination. When a new message comes, the world checks if it is a termination message. For termination messages, the algorithm program executes a basic echo algorithm. We will not use this approach for the next section because we need to restart rabbitmq. Therefore we use an easier approach (flooding exit message in the network) for the next section.

2.3 algorithm.py

For wave messages, the world calls the process_msg function. In this function, the system id of the node is added to got_msg_from list, and the message does not have

any other impact if the message has come from itself. About other messages from other nodes, this function calls the echo_extinction function that the algorithm implemented in this function. Whenever got_msg_from list contains all neighbors that sent wave messages with the current leader id and current leader round, it is time to report the subtree size to the parent. The parent has found in echo_extinction function based on the algorithm. The node constructs a new wave message containing its subtree's size and sends it to its parent. If the node does not have any parent, it means that there was no bigger round or id. The node checks the size of the subtree. If the subtree size is equal to the size of the network, the node starts termination echo; otherwise, it starts a new round and resets all received data.
In echo_extinction function, the algorithm implemented that can be found in [2]. Therefore I just write some important points. Whenever the id and round of the message are equal to id and round of node, this message is considered as the size reporting message, and the subtree size of the message appends to subtree_size list, and the sender system id appends to got_msg_format list. Whenever the id or round of the message is bigger than the node id or round, the node sets the sender as its parent, resets all data, and continues the new wave.
the result of this section is in the output directory in the last run directory.

3 Message Complexity in Different Networks

3.1 message complexity of algorithm

All nodes are initiators; therefore, we have N wave in the first round. Each wave uses at most $2 * E$ messages. Then we have $O(N * E)$ messages in the first round. In the next round, we have M initiators that $M \leq N$. Therefore after the T round, we used $O(T * N * E)$ messages and definitely $T \leq N$, so we have at most $O(N * E)$ messages.

3.2 different network simulation

As discussed in the previous section, we use an easier approach (flooding exit messages in the network). I added another attribute to the world class (number_of_messages) to keep the number of messages that a node received, and it writes this number when it wants to terminate. I wrote a new python program to create different graph types with different numbers of nodes(create_graph.py). Also, I wrote a shell script to run those graphs, and after ending each run, it restarts the rabbitmq service. I ran the simulation five times for the complete graph, star graph, path graph, and ring topology. All these simulations were run for the different number of nodes between 5 to 45. Also, I simulated the given tree (input_.in) five times.
To evaluate the message complexity of each type of graph, I wrote another program (evaluation.py). In this program, I construct a dictionary for each type of network, and

after that, it starts to read stdout files for each run. It adds the number of messages for every node to the total number of messages and appends the total number of messages for that run to the corresponding list of that type of dictionary. Finally, this program draws different diagrams and writes the results in a file.
All results are available in results directory in section_2 directory.

3.3 Results

In this section we discuss the results of the simulations.

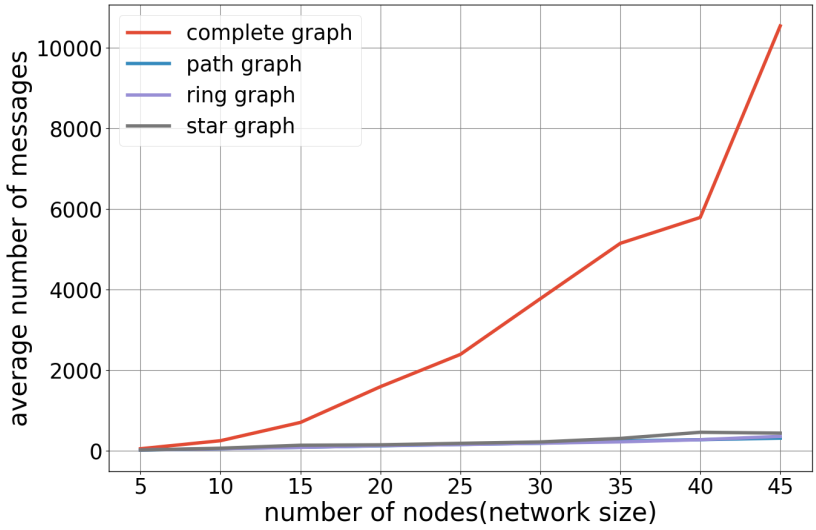

Figure 1: number of messages per number of nodes for different networks

Figure 1. shows the difference between types of popular networks in message complexity. In a complete graph, we can see that the number of messages is increased dramatically, but in other networks, it is like the linear function.
Figure 2. shows the number of messages for every network type individually. I inferred that the best graph suitable for this algorithm is the path graph because the message complexity has a linear function with the number of nodes.
For the first given tree graph (input_001.in) we the average number of messages is equal to 77 messages. For the first given tree graph (input_002.in) we the average number of messages is equal to 48 messages. If we have packet loss (with probability 0.01)for one edge we have 61 average messages. We can say that if we have packet loss in the system the number of messages will be increased.

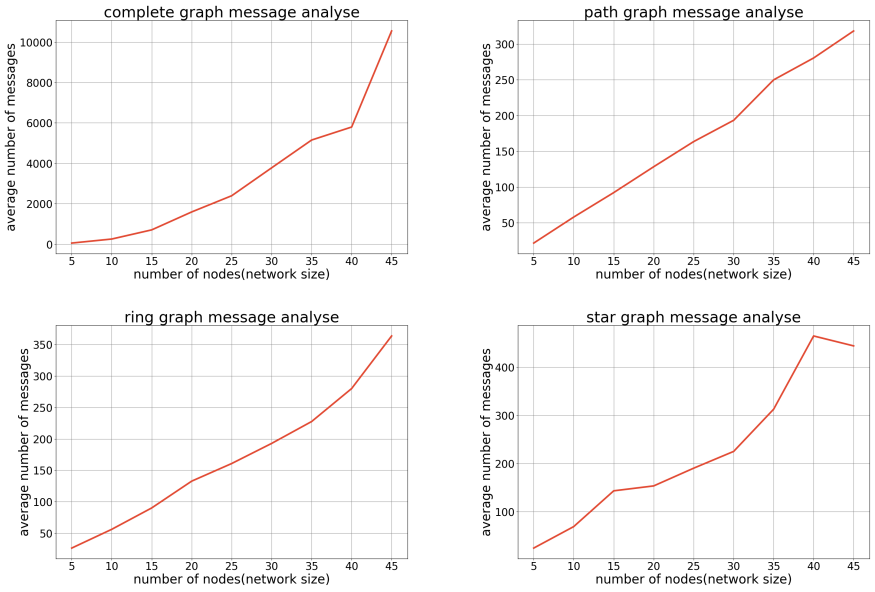

Figure 2: number of messages per number of nodes for different networks

4 Novel Approaches Implementation

I implemented [1] for this section. You can find the algorithm with details in [1], and in this section, I explain the implementation of the algorithm.
The paper said that every node broadcasts its CRQ messages, and all nodes can communicate with each other. Thus, I use the only complete graph in this section.

4.1 create_config.py

At first, we need to have a configuration file to find every node priority for the first phase. In the paper, they said that we have to find the highest priority, but I assumed that we already knew the nodes' priority. Therefore, I wrote a program (create_config.py) to create configuration files. In this program, the priority of each node is selected randomly (a number between 1 to 3). A high number means a high priority. It constructs this configuration file for the number of nodes between 5 to 45.

4.2 runner.py

To simulate with configuration files, I added a new argument for the runner process and node process to give this file to the program. In addition, we have to use SimulatorFullView because all nodes have to know all other nodes. Thus we create a node process with SimulatorFullView argument.

4.3 node.py

Like the first section, after starting the node process, all nodes are initiators, and they start the first phase implemented in the world file.

4.4 world.py

In this section, we use SimulatorFullView. When the node calls the start_first_phase method of the world, the program finds the node's priority to see if the node is filtered and is passed the first phase. Next, it finds the high-priority nodes and makes a list of these nodes. If the node has a high priority, it starts the second phase by calling start_second_phase method. In the second phase, the node constructs a CRQ message and sends it to all nodes.
Whenever a node receives a new message, it calls the process_msg from the algorithm program.

4.5 algorithm.py

We have 5 different types of messages.

- **CRQ** : When the received messages are in this type, the node has to decide to vote the source, so it calls handle_crq_message function. In this function, if the node has already voted, it sends a vote message with 0 counts to the source. If the message came from itself, the node sends a vote message with one count to itself. If the node is active, it sends a vote message width 0 counts, and otherwise, it votes the source by sending a vote message with 1 count.

- **vote** : When a node receives this kind of message, count the vote to its votes. Whenever a node gets vote messages from all other nodes, it sends CHANGE message, including its number of votes to other active nodes.

- **CHANGE** : When the received messages are in this type, the node has to decide to accept the source as a leader, so it calls handle_change_message function. In this function, if the number of source votes is bigger than its votes, or they have the same votes, but the system id of the source is bigger than its system id, it sends an ACK message to the source.

- **ACK** : When the received messages are in this type, it appends the source to the list of nodes that sent the ACKmessage to this node. Whenever a node gets ACK message from all active nodes, it expresses itself as a leader and starts the last phase by sending a CAM message to all nodes.

- **CAM** : When the received messages are in this type, the node writes the results and number of messages and terminates.

4.6 results

I ran the simulation for various nodes from 5 to 45 and wrote a shell script like the second section to run. For evaluation, I wrote a python code like the second section. Results are in the results directory in the corresponding folder.

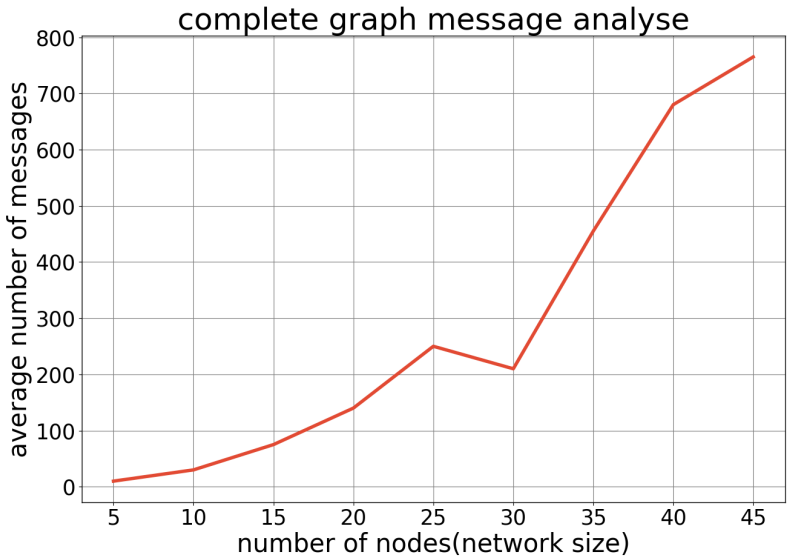

Figure 3: number of messages per number of nodes for 3-phase election algorithm

We can see that the number of messages have a exponential relation with number of nodes. The message complexity is $O(N^2)$.

References

[1] Pushya Chaparala, Aparna Rajesh Atmakuri, and S Siva Sankara Rao. 3-phase leader election algorithm for distributed systems. In *2019 3rd International Conference on Computing Methodologies and Communication (ICCMC)*, pages 898–904. IEEE, 2019. pages 7

[2] Wan Fokkink. *Distributed algorithms: an intuitive approach*. MIT Press, 2018. pages 4, 5