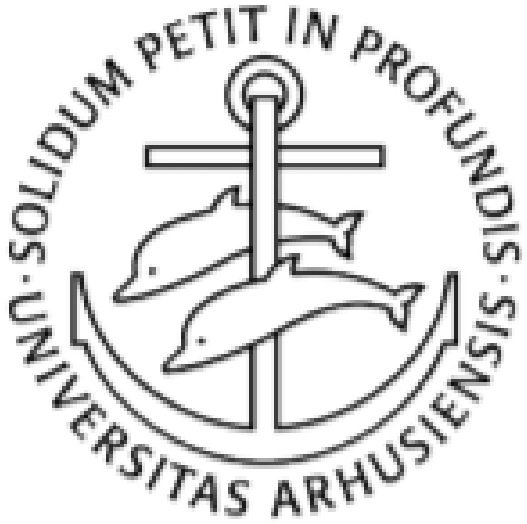
Distribute and Pervasive System
Implementation Bully Algorithm : Group 7



IMPLEMENTATION
BULLY ALGORITHM

DISTRIBUTED AND PERVASIVE SYSTEM

AARHUS UNIVERSITY

SUBMITTED: 13.03.2022

Name:	Studie program:	Au id.:
Fernando Pannullo	Computer Engineering	au703559
Haoran Li	Computer Engineering	au703092
Weize Kong	Computer Engineering	au703116
Navid Eftekhar Ardebili	Computer Engineering	au688042

Contents

Introduction 1

Leader(coordinator) Election is an important problem in distributed computing systems. A distributed system is a collection of autonomous nodes which can communicate with each other. Each node that stores a copy of the database is called a replica[3]. In order to ensure that all the data ends up on all the replicas, leader election is introduced, which is helpful for data synchronization. For example, every write operation to the database needs to be processed by the leader and leader then synchronize data to its followers so that all the nodes contain the same data, that is, read operation can be realized either from leader or followers.

Leader election runs when a system starts initially or we need to add or remove nodes from the system. It can be also used when the failure of the leader is detected by the followers. We focus more on the *init* system part. Thus, when the nodes in the system are initially turned on, they will automatically elect a leader.

Bully Election is one of the leader election algorithms[1]. "Bully" means node with highest ID forces the nodes with smaller ID to accept it as a leader. This algorithm is relatively simple and easy to implement but it involves a high number of messages exchanged between processes. Therefore, heavy traffic might increase time and computation cost, which is not energy saving. Nowadays, IoT technology is soaring and IoT devices can also be considered a distributed system. But they are usually constrained devices which may not stand for such heavy traffic of bully election. So optimizing bully elections is necessary both for the future and for energy savings.

In this assignment, we initially explain, implement and test the Bully Algorithm and discuss its drawbacks and come up with an improved bully algorithm, which performs better than the previous one.

Theory 2

2.1 Bully Algorithm

In this assignment, we've finished implementing the Bully Algorithm and improved the Bully Algorithm. Here are the descriptions of two algorithms.

2.1.1 Bully algorithm

Bully election was first proposed by Hector Garcia-Molina in 1982[2]. When the system is initialized, the bully election runs and selects one and only one leader with the highest ID. Obviously, the precondition is that each process has a unique ID and nodes know each other's ID so that they can compare thus find the highest. In addition, nodes communicate with each other by sending 3 different types of messages:

1. Election: message to announce election
2. OK: response message, which means "OK, I am alive and have a higher ID, so let me take over the election"
3. Coordinator: Message to announce ID of new coordinator

The procedure of bully election is shown as follows:

- node i tries to contact all nodes with higher ID by sending Election message
  - If no OK message returns after the time limit
    - node i sends Coordinator message to nodes with lower ID because node i guarantees that those nodes with higher ID have failed
  - If OK messages returns
    - node i is muted and awaits a Coordinator message
  - If node i receives Coordinator message
    - node i treats the sends as a leader

2.1.2 Improved bully algorithm

From the algorithm above, we can see the worst situation is that the node with least ID starts the election. Therefore, an expected n^2 number of messages (with n the number of nodes within a network) will be sent.

We aim to modify the bully algorithm, in order to reduce the number of messages needed to be exchanged to find the coordinator. If a node with lowest ID starts an election, it only sends the Election message to the node

with highest ID and awaits for response. If there is no response within the time limit, it then sends an Election message to the node with the second highest ID and so on, until the OK message returns.

In short, the improved bully election can reduce messages needed to be exchanged, because it only sends one Election message to the node with the highest ID. But it also increases time cost since nodes with higher ID might fail at the moment. Time cost and message complexity need to trade off, we will discuss in the following chapter.

Implementation 3

Implementation overview

3.1 Tools and Configurations

3.1.1 Docker container

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

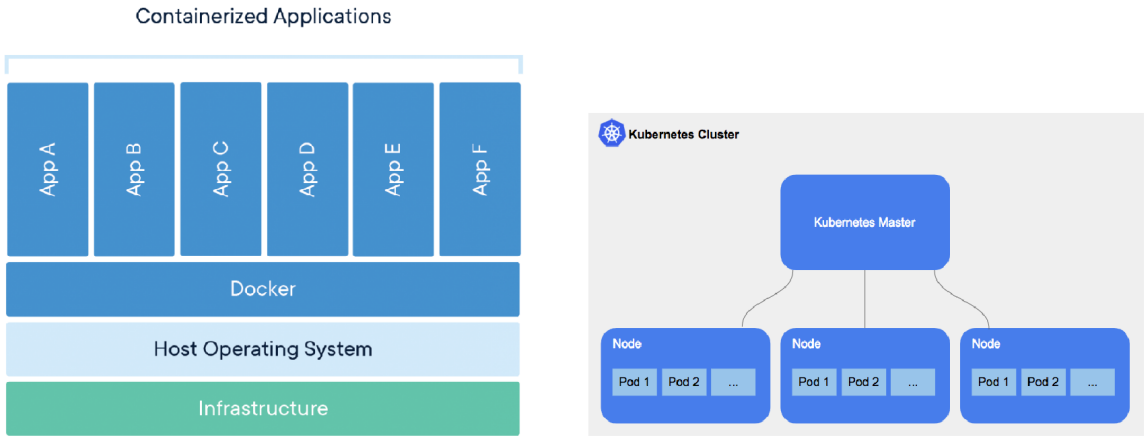


Figure 3.1. *Containerized Apps and Kubernetes managing*

In our implementation, we leverage docker images as different nodes in a distributed system. According to the principle of distributed systems, each node should be at an equal level. Unlike Client/Server architecture, distributed systems do not have a central server, but let these nodes communicate with each other and exchange messages. So, in order to implement the bully election algorithm, and also the improved one, we only need to code once, then create several docker images. Each docker image is a node in a distributed system and has its

own IP address. We can use Docker commands to build such images and boot them up. When several docker images are booting up, they will execute commands and form a cluster.

To successfully create a docker image, we need a configuration file called *Docker file*. In this file, we need to specify the docker image name, booting commands and port we want to expose. In our case, we use docker image with python environment, install all package dependencies and run $main\_server.py$ when the system is booting up. We also exposed port 5000 for others to access.

```
FROM python:3.7

RUN mkdir /app
WORKDIR /app
ADD . /app/
RUN pip install -r requirements.txt

EXPOSE 5000
CMD ["python", "/app/python/main_server.py"]
```

The script below builds a docker image from $Dockerfile\_host\_0$ and labeled leader-election-python-0. In our task, since we need to create multiple docker images, we can use bash scripts instead of typing several docker commands each time. We will use Kubernetes to manage all the Docker containers, so we don't need to boot them up manually.

```
#!/bin/bash
docker build -f Dockerfile_host_0 -t leader-election-python-0:latest .
```

3.1.2 Kubernetes

Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications. It groups containers that make up an application into logical units for easy management and discovery.

In our case, we use the built-in version of Kubernetes inside docker desktop. To achieve this, we need to switch on Kubernetes in docker desktop options and reload the app. Like dockers, we also need configuration file for k8s to work. In this *.yaml* file, we can specify the network protocol and port exposed, the number of hosts, type of hosts and some environment variables passing to different hosts. Below is part of the configuration file which specifies the port configuration for leader-election-python. $targetPort$ should be the same as the port exposed in the docker file. Port is exposed for outside, like in our host machine, and nodePort is for internet connections.

```
spec:
  selector:
    app: leader-election-python
  ports:
    - name: tcp0
      protocol: TCP
      nodePort: 30000
      port: 5010
      targetPort: 5000
```

After configuration of both docker and kubernetes, we can safely launch the kubernetes cluster and observe the cluster status by these bash commands.

```
#!/bin/bash
kubectl apply -f deployment.yaml
kubectl get pods
```

3.2 Core logic of bully algorithm

As mentioned before, each node in this cluster has the same program to run. We implemented this core logic of the Bully Election Algorithm and also improved one, which drastically decreased the execution time and message payload. In order to exchange information easily, we leverage a RESTful tool, Flask in Python, to be our main frame. Each node will act as a HTTP server and respond to others. This is done in $main\_server.py$. Besides, all variables and related logic of the bully algorithm is implemented in $bully\_logic.py$ and $bully\_logic\_improved.py$. In next subsections we will talk about different phases in the bully algorithm.

3.2.1 Initialization

Each node has the same program to run, but actually, to initialize and launch a new election, we will specify one node to start. We do this through the environment variable $MUTEX$. This variable can be configured through a *.yaml* file for Kubernetes. If this node's port matches with $MUTEX$, then he will be the starter and do $preamble$. While others just start waiting and listening.

According to the definition of the bully algorithm, each node has full information about all other nodes. In $preamble$, this node will generate a unique and random ID for himself and others. These IDs cannot be the same because we need to use them as a credential to elect a node with the largest ID as the coordinator. We used current time plus a random integer as a new ID. Then, this starter node will send these IDs to corresponding nodes, and this is called $register$. Basically, assign a unique ID for each node.

```
service_register_status= None

self.hosts_ports= self.define_ports()

for host in self.hosts_ports:
   candidate= self.generate_node_id()
   while candidate in self.ids_nodes:
      candidate = self.generate_node_id()

   self.ids_nodes.append(candidate)

for host in self.hosts_ports:
   id_num= 0
   for port_id in self.hosts_ports:
      service_register_status = self.register_service(host, port_id,self.ids_nodes[id_num], id_num)
      id_num += 1

self.start()
```

3.2.2 Election

After initialization, the starter will launch an election. In the original bully version, this node will find all other nodes with higher IDs, and send them messages $redirect$. This message also contains the higher IDs that each receiver needs to redirect. For example, if we have five nodes from node 1 to node 5. If node 2 is a starter. Node 2 will send {redirect: 4, 5} to node 3, {redirect: 5} to node 4, and {redirect: null} to node 5.

This node will act differently according to the response. If all nodes respond, fail or start, it itself has the largest ID. Then he will stop and go to the $announce$ stage. Once one of the nodes responds to the starter correctly, it will stop working and that node will take over his responsibility. This job will act the same as a starter. Finally, the node with the largest ID will win this election and enter $announce$. The pseudo code is shown below.

```
Higer_ID_list = get_higher_IDs()
response_list = []
if Higher_ID_list:
   for ID in Higher_ID_list:
      message = generate_message(ID)
      response = send_message_to(ID, message)
      response_list.append(response)
else:
   announce()

if '200' not in response_list:
   announce()
```

3.2.3 Announce

In this stage, the responsibility has been taken over by the nodes with the largest ID, and this node has been elected as a new coordinator in this cluster. Since it has all the information of other nodes, it will announce to them that the current election round has finished, and itself is the new coordinator. The pseudo-code is shown below.

```
ID_list -= self.ID
for ID in ID_list:
   send_message_to(ID, self.ID)
end_election()
```

3.2.4 Improved version

In the original version of the bully algorithm, one node needs to inform or send a $redirect$ message to all the nodes with higher IDs. This mechanism is to ensure that it will not miss any nodes with higher IDs. But in this way, too many messages are sent back and forth. The total amount of message transmission is really high, which can have an impact on the performance especially when the number of nodes is large. So there comes an improved version of Bully Algorithm, and in this algorithm, one node will only send message to the node with the largest ID. The act of sorting the largest ID machines from a list of candidates is made from a Bubble-Sort Algorithm. Once this node fails, the starter will send a message to the node with the second largest ID. If all nodes with higher ID are down, then starter is coordinator itself. If any of node receives messages from others, then it will be the coordinator and announce it to others. In this way, the amount of messages will decrease drastically and have a better performance. Pseudo code is shown below.

```
1 sort(ID_list)
2 for ID in ID_list:
3    respond = send_message_to(ID)
4    if respond == 200:
5       break
6    else:
7       continue
8 wait_for_announce()
```

Experimentation of classic Bully Algorithm 4

The experiments in this project are basically of two types: will be described separately to give a better explanation of each, even use at the base the same code, for give different interpretation of the results and explanation. The main idea of the test conditions is to increase the computational and network communication in setup steps, with increasing the number of participants of the election (5 - 10 - 20 containers used). The metrics are essentially also three as time to complete the election, total bytes transferred from all the system, e.g. not only the containers involved in the election and the total of the number messages exchanged with the same range of domain as the total bytes. Given all of these variables of functioning we can extract information about the quality of this algorithm can offer in reality and cons. All the tests are repeated 3 times and we will give the near mean values.

All the tests are made from a desktop-PC in Windows environment equipped with AMD FX-8370 with 32GB of RAM. For managing the container is used *Docker Desktop* that allows the use of docker in Windows environment. Kubernetes service is provided also, from an add-in form Docker Desktop. The start of the algorithm is delayed by 15 seconds, to admit all the containers to reach a ready state and the same internal workload when the system is deployed. This time also ensures that also with 20 containers, (that is the worst case) to load from Kubernetes are fully ready.

4.1 Purpose

In this experimentation we test the real capability of the pure *bully algorithm* in a real case in a setup described before in chapter 3. These measurements are valuable to describe the algorithm not only in speed terms but also for trying to predict the behavior in a real implementation where the contenders for the election can reach the 100 hosts or more. With the pure algorithm we have extracted the time of computational to establish a *leader* from a single common point to stop after a successful sending of election messages. For the intention to experiment with the possible speed improvements in this particular algorithm, we have chosen to use threading to check for possible improvements after parallelizing the function *go_deep*, e.g. when elections are redirected from the first one.

4.2 Code and data

The techniques to extract the data are implemented all in the class *logic*, that holds, as the name suggests, the logic of the algorithm. With only adding two *libraries* offered from python environment as *time*, specifically

with *perf_counter* function that give a time-stamp over three cipher of precision and finally *sys* for extract the weights in byte of a general object. The register that holds this information, is not distributed or shared in the system during the election but each machine keeps his own information, to release after the election with a GET request to all hosts in the code below. This is for not false the measurement of the data transferred and time occurred. The information are saved into a dictionary type variable called *metrics* that is composed from a elements *time_start*, *time_finish*, *size* and *messages*, available all the time during the election for modification only from the node himself.

```
details = []
   for each_port in self.hosts_ports:
      if each_port != self.port_local:
         url = self.url_local + str(each_port) + '/performance'
         data = requests.get(url).json()
         if data['time'] != 0:
            self.metrics['time_finish'] = self.metrics['time_finish'] - data['time']
         self.metrics['size'] += data['size']
         self.metrics['messages'] += data['messages']
      else:
         if self.metrics['time_start'] != 0:
            self.metrics['time_finish'] = self.metrics['time_finish'] - self.metrics['time_start']
```

4.3 Metrics

The metrics acquired as anticipated before are 3: The first is the time to accomplish the algorithm that starts, after the function has finished to send all information needed as every container knows, all information of each other, ID, port of communication, engagement in a election, and the id of the leader, if there is one. The end-time is taken when all messages of the winner announcement are sended and confirmed by the other part of the reception. The other one is the size and the number of the messages, that are functionally correlated with the methodology described before. Every request sended of type PUT and POST metrics are stored in the same container, instead a POST is evaluated and stored to the responder.

4.4 Result

The results are described in the table that describe 3 tests related to the quantity of container, used using the described metrics. All information are derived from a log file, automatically produced from HTML result page on http:\\localhost:<port_that_correspond_to_a_container>, but at the winner-election page a download link provide a log file directly from the machine.

Table 4.1. Result of the benchmark.

Classic Bully Algorithm			
	5 Containers	10 Containers	20 Containers
Time elapsed in seconds	0.038	0.107	0.262
Total of bytes transferred	1984	27032	44392
Total of messages transferred	8	109	179

4.5 Discussion

As we can see in the table of the result the time elapsed correlated to the 5 containers test and doubled test types of 10 and 20 are very close to linear. Basically at a doubled increase of the containers, there is an increase of the time elapsed by about 2.8 times from the test, with 5 to 10 containers and about 2.4 times from 10 to 20. The amount of bytes and messages transferred are 13.6 times more, but from 10 to 20 containers the difference is only 1.6 times, also obviously for the quantity of the messages. So there is a slight flattening on the curve that represents the increments of the metrics of data transfer. Seems that this algorithm tends to give some better performance in terms of speed, with a high number of containers examining the parameters growth increase within the tests. After a quick view the trend has slightly more steepness as shown in Figure 4.1 because the absolute values are bigger, but considering the ratio in reality is lowered from 2.8 to 2.4 times slower.

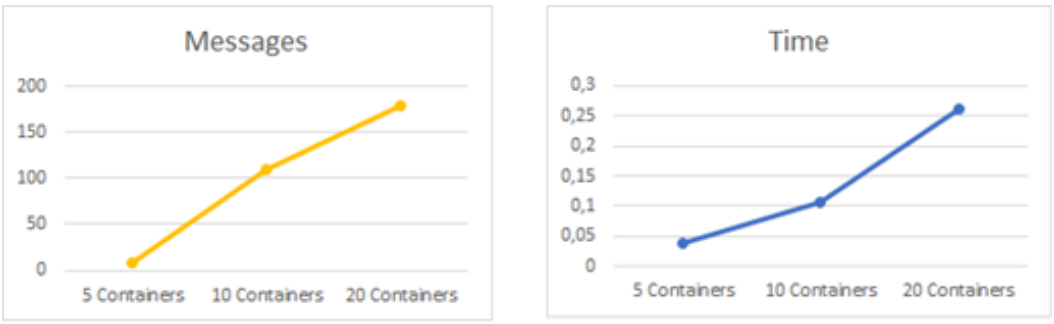


Figure 4.1. Messages and Time-elapsing representation through the tests

Experimentation of improved Bully Algorithm 5

This experiment is pretty similar to the *classic* one, because it uses the same interface to communicate to the other containers and the same logic.The environment also is the same, as the *classic* Bully Algorithm tests.

5.1 Purpose

This purpose also has a similarity to the *classic* one. With the only difference is that the parallelizing of the function *go_deep*, here has significantly less use than the *classic* one. So giving this, we can extract some information about also a benefit using threading, but also if there is some benefit with using less messages communication, paying the price for more computation for ordering an array of candidates.

5.2 Code and data

The techniques to extract the data are implemented all in the class *logic*, that holds, as the name suggests, the logic of the algorithm. The extraction, procedures of the metrics and composition of the data are exactly the same of *classic* version of the algorithm, in chapter 4.2.

5.3 Metrics

The metrics acquired as anticipated before are 3 as the previous experiment with the same characteristics of extraction and quality, described in chapter 4.3.

5.4 Result

The results are described in the table that describe 3 tests, related to the quantity of container used using the described metrics. All information are derived from a log file automatically produced from HTML result page, like described in chapter 4.4.

5.5 Discussion

As we can see in the table of the result, the time elapsed correlated to the 5 containers test and doubled test types of 10 and 20 are very close to linear. Basically at a doubled increase of the containers there is an increase of the time elapsed by about 1.9 times from the test with 5 to 10 containers and about 2.3 times from 10 to 20.

Table 5.1. Result of the benchmark.

Classic Bully Algorithm			
	5 Containers	10 Containers	20 Containers
Time elapsed in seconds	0.028	0.053	0.124
Total of bytes transferred	1240	2480	4960
Total of messages transferred	5	10	20

The amount of bytes and messages transferred are exactly doubled, for each increment of the docker machines in the tests, so perfectly linear. The same trend obviously is for the quantity of the messages. This happens because in all the tests all containers work properly with no failures. But the algorithm has a structure to add only one message for each failure found, so is a good estimation also in reality and estimation of boundaries. Seems that this algorithm does not tend to give some better performance in terms of speed, with a high number of containers examining the parameters growth increase within the tests. After a quick view the trend has slightly more steepness as shown in Figure 5.1 in terms of absolute value of time, but the ratio in reality is raised from 1.9 to 2.3 times slower. Last note, the messages don't increase a lot in changing the cases, so this specular consideration in terms of time can be mitigated from this feature.

Figure 5.1. Messages and Time-elapsing representation through the tests

Comparison and Conclusion 6

The execution of the tests shows all the true behavior behind a system that simulates a real computing distribution, with the use of RESTful tool that can be easily scaled to more complicated applications. Indeed, we have to consider that in a real implementation the location of the hosts can be very distanced and maybe not in the same network as in this case.

Table 6.1. Result of the benchmark with direct comparison

Classic Bully Algorithm/Improved Bully Algorithm			
	5 Containers	10 Containers	20 Containers
Time elapsed in seconds	0.038/0.028	0.107/0.053	0.262/0.124
Total of bytes transferred	1984/1240	27032/2480	44392/4960
Total of messages transferred	8/5	109/10	179/20

A comparison of these two types of this algorithm shows a clear benefit to use the *improved version*, in all of the aspects considered. The time advantage is very low for a little number of containers, but is about the halftime need for the 10 and 20 containers case in relation to the classic one adding on top that ,there is a thousandth improvement on the 20's case. This can lead to thinking that with more containers or hosts connected the improvement can be more clear.

For what concern messages and bytes transferred the improvement are clear with more than 10 containers the advantage is astonishing. Indeed in the case of less than 5 the messages are lowered for a little amount but the rest of the test shows a less inter-exchange of messages reduced by 10 times. This can contribute for less computation and managing of the packets, for all machines involved in the election, with for a leveraged counterpart of computing load in sorting. This can be a trade-off to consider when we have a system with some constraint about the channel of data-transmission.

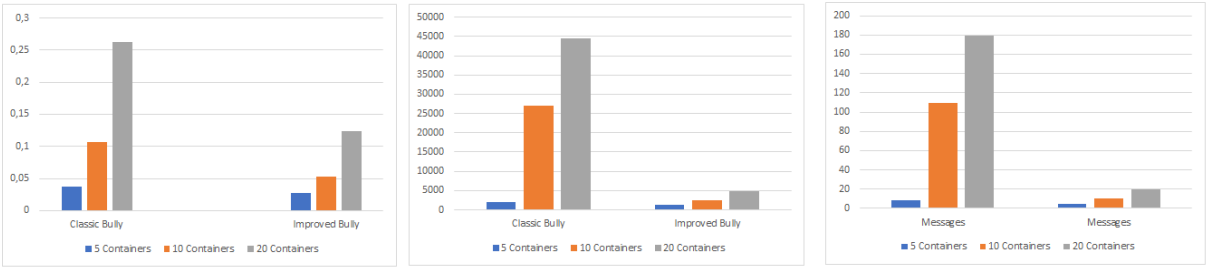


Figure 6.1. Comparison of the results from the left: Time, weights and number of messages

In the future the improved algorithm can be improved by inserting a sorting algorithm different, from Bubble Sort used in this example, that has wide boundaries of difficulty as worst and average n^2 complexity. The choice can be made from examples: Insertion Sort, Merge Sort, Quicksort, Counting Sort and so on that have different boundaries of complexity that can be adapted for different use cases.

Bibliography

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

[2] Garcia-Molina. "Elections in a Distributed Computing System". In: *IEEE Transactions on Computers* C-31.1 (1982), pp. 48–59. DOI: 10.1109/TC.1982.1675885.

[3] Martin Kleppmann. *Designing Data-Intensive Applications*. O'Reilly, 2017. ISBN: 978-1-4493-7332-0.