Tribhuvan University
Institute of Engineering
Pulchowk Campus

Distributed Systems
Lab 6:

Implementation of a Distributed Election Algorithm

Submitted by:

Aayush Lamichhane
(075BCT004)

Bishal Katuwal
(075BCT028)

Submitted to:
Department of Electronics and Computer Engineering,
Pulchowk Campus

Submitted on:
August 9, 2022

Title

Implementation of Distributed Election Algorithm

Basic Theory

Need of Distributed Elections
Many applications within distributed networks require some sort of coordinator and/or leader to work. One of the problems with these leader-dependent algorithms is dependency on a single leader that introduces a single point of failure in the entire network. If the leader is down, the rest of the system will either go down or as well, or be faulty in other ways (give unpredictable results, have out-of-date data, etc.).
To solve this problem and make distributed systems more stable and reliable, nodes can take over the leader role when the current leader of the system fails. Determining which node takes over in case of a leader failure is done through a so-called leader election algorithm.
Different topologies (designs) of distributed networks have different leader election algorithms that work best on them. This report focuses on fully connected networks, as these types of networks resemble the internet the most, with any machine being able to connect to any other machine. One of the leader election algorithms that works well on fully connected networks is the Bully algorithm.

Bully Algorithm
In this algorithm, every node within the network has an ID and a list of all other nodes in the network. Nodes can detect other nodes failing, and can initiate an election of a new leader if necessary.
The algorithm itself works as follows: If a node detects leader failure, it will send an election message to all nodes with a higher ID and will wait for a response to these messages. If there's no response after a certain amount of time, the node will promote itself to the leader. If there is a response, it means there are nodes with higher IDs up and running, resulting in the current node to wait until one of the nodes with a higher ID elects itself as leader. If the node detecting leader failure is the node with the highest ID, it will not send any election messages and immediately promote itself to the new leader of the network. Nodes receiving an election message will send back an alive message and will also start the leader election process themselves, sending out election messages to nodes with higher ID. Once a node promotes itself to the new leader of the network, it will notify all other nodes of this, finishing the leader election process.
This can be simplified as:

- Each node has a unique ID.
- Each node communicates with each other and broadcasts their IDs.
- The node which has the highest ID becomes the Leader.

Messages in Bully Algorithm:
Bully algorithm has following message types:
Election Message: Sent to announce the election.
Answer (Alive) Message: Responds to the Election message.
Coordinator (Victory) Message: Sent by the winner of the election to announce victory.

Implementation
Assumption:
The algorithm assumes that:
- The system is synchronous.
- Processes may fail at any time, including during execution of the algorithm.
- A process fails by stopping and returns from failure by restarting.
- There is a failure detector which detects failed processes.
- Message delivery between processes is reliable.
- Each process knows its own process id and address, and that of every other process.

Algorithm:

When a process P recovers from failure, or the failure detector indicates that the current coordinator has failed, P performs the following actions:

- If P has the highest process ID, it sends a Victory message to all other processes and becomes the new Coordinator. Otherwise, P broadcasts an Election message to all other processes with higher process IDs than itself.
- If P receives no Answer after sending an Election message, then it broadcasts a Victory message to all other processes and becomes the Coordinator.
- If P receives an Answer from a process with a higher ID, it sends no further messages for this election and waits for a Victory message. (If there is no Victory message after a period of time, it restarts the process at the beginning.)
- If P receives an Election message from another process with a lower ID it sends an Answer message back and if it has not already started an election, it starts the election process at the beginning, by sending an Election message to higher-numbered processes.
- If P receives a Coordinator message, it treats the sender as the coordinator.

Lab 6: Election Algorithm

We use Consul as the service registry. It manages the nodes that are active and provides information about other nodes that are currently active.

Installation: `sudo pacman -S consul`

start: `consul agent -dev`

Consul runs at `http://localhost:8500/v1/agent/service/register`

Each node stores some information related to the coordinator in the Bully object

```
class Bully:

    def __init__(self, node_name, node_id, port_number, election=False, coordinator=False):
        self.node_name = node_name # what is my name?
        self.node_id = node_id # who am i?
        self.port = port_number # what is my address
        self.election = election # is the election going on?
        self.coordinator = coordinator # who is the coordinator
```

The process will check health of the coordinator process every 60s

```
def check_coordinator_health():
    threading.Timer(60.0, check_coordinator_health).start()
    health = check_health_of_the_service(bully.coordinator)
    if health == 'crashed':
        init()
    else:
        print('Coordinator is alive')
```

If the coordinator has crashed, `init()` is called.

```
def init(wait=True):
    if service_register_status == 200:
        ports_of_all_nodes = get_ports_of_nodes()
        del ports_of_all_nodes[node_name]

        # exchange node details with each node
        node_details = get_details(ports_of_all_nodes)

        if wait:
            timeout = random.randint(5, 15)
            time.sleep(timeout)
            print('timeouting in %s seconds' % timeout)

        # checks if there is an election on going
        election_ready = ready_for_election(ports_of_all_nodes, bully.election, bully.coordinator)
        if election_ready or not wait:
            print('Starting election in: %s' % node_name)
            bully.election = True
            higher_nodes_array = get_higher_nodes(node_details, node_id)
            print('higher node array', higher_nodes_array)
            if len(higher_nodes_array) == 0:
                bully.coordinator = True
                bully.election = False
                announce(node_name)
                print('Coordinator is : %s' % node_name)
                print('**********End of election**********************')
            else:
                election(higher_nodes_array, node_id)
    else:
        print('Service registration is not successful')
```

The init method checks if the process executing it is properly registered to the service register. Then it gets the details of all nodes and then waits for random time depending on if the wait parameter is True. Then, it checks if the node is ready for election using details from all other nodes and its own status. If it is election ready then the election is started.

During election, the nodes with higher priority are found out and each of the higher nodes is sent the node's id saying that the election is being conducted. Each node has an Endpoint for receiving this message

```
@app.route('/proxy', methods=['POST'])
def proxy():
    with counter.get_lock():
        counter.value += 1
        unique_count = counter.value

    url = 'http://localhost:%s/response' % port_number
    if unique_count == 1:
        data = request.get_json()
        requests.post(url, json=data)

    return jsonify({'Response': 'OK'}), 200
```

This endpoint is a helper to forward only one request to another endpoint (/response) using locks

```
@app.route('/response', methods=['POST'])
def response_node():
    data = request.get_json()
    incoming_node_id = data['node_id']
    self_node_id = bully.node_id
    if self_node_id > incoming_node_id:
        threading.Thread(target=init, args=[False]).start()
        bully.election = False
    return jsonify({'Response': 'OK'}), 200
```

This endpoint gets the id of the node announcing the election and if the node has higher id accepts the message otherwise starts its own init process(which will get node statuses and then restart the election)

Each process has a endpoint for receiving the announcement of the coordinator

```
@app.route('/announce', methods=['POST'])
def announce_coordinator():
    data = request.get_json()
    coordinator = data['coordinator']
    bully.coordinator = coordinator
    print('Coordinator is %s ' % coordinator)
    return jsonify({'response': 'OK'}), 200
```

```python
import time
import json
import requests
from random import randint


def generate_node_id():
    millis = int(round(time.time() * 1000))
    node_id = millis + randint(800000000000, 900000000000)
    return node_id


# This method is used to register the service in the service registry
def register_service(name, port, node_id):
    url = "http://localhost:8500/v1/agent/service/register"
    data = {
        "Name": name,
        "ID": str(node_id),
        "port": port,
        "check": {
            "name": "Check Counter health %s" % port,
            "tcp": "localhost:%s" % port,
            "interval": "10s",
            "timeout": "1s"
        }
    }
    put_request = requests.put(url, json=data)
    return put_request.status_code


def check_health_of_the_service(service):
    print('Checking health of the %s' % service)
    url = 'http://localhost:8500/v1/agent/health/service/name/%s' % service
    response = requests.get(url)
    response_content = json.loads(response.text)
    aggregated_state = response_content[0]['AggregatedStatus']
    service_status = aggregated_state
    if response.status_code == 503 and aggregated_state == 'critical':
        service_status = 'crashed'
    print('Service status: %s' % service_status)
    return service_status


# get ports of all the registered nodes from the service registry
def get_ports_of_nodes():
    ports_dict = {}
    response = requests.get('http://127.0.0.1:8500/v1/agent/services')
    nodes = json.loads(response.text)
    for each_service in nodes:
        service = nodes[each_service]['Service']
        status = nodes[each_service]['Port']
        key = service
        value = status
        ports_dict[key] = value
    return ports_dict


def get_higher_nodes(node_details, node_id):
    higher_node_array = []
    for each in node_details:
        if each['node_id'] > node_id:
            higher_node_array.append(each['port'])
    return higher_node_array


# this method is used to send the higher node id to the proxy
def election(higher_nodes_array, node_id):
    status_code_array = []
    for each_port in higher_nodes_array:
        url = 'http://localhost:%s/proxy' % each_port
        data = {
            "node_id": node_id
        }
        post_response = requests.post(url, json=data)
        status_code_array.append(post_response.status_code)
    if 200 in status_code_array:
        return 200


# this method returns if the cluster is ready for the election
def ready_for_election(ports_of_all_nodes, self_election, self_coordinator):
    coordinator_array = []
    election_array = []
    node_details = get_details(ports_of_all_nodes)

    for each_node in node_details:
        coordinator_array.append(each_node['coordinator'])
        election_array.append(each_node['election'])
    coordinator_array.append(self_coordinator)
    election_array.append(self_election)

    if True in election_array or True in coordinator_array:
        return False
    else:
        return True


# this method is used to get the details of all the nodes by syncing with each node by calling each nodes' API.
def get_details(ports_of_all_nodes):
    node_details = []
    for each_node in ports_of_all_nodes:
        url = 'http://localhost:%s/nodeDetails' % ports_of_all_nodes[each_node]
        data = requests.get(url)
        node_details.append(data.json())
    return node_details


# this method is used to announce that it is the master to the other nodes.
def announce(coordinator):
    all_nodes = get_ports_of_nodes()
    data = {
        'coordinator': coordinator
    }
    for each_node in all_nodes:
        url = 'http://localhost:%s/announce' % all_nodes[each_node]
        print(url)
        requests.post(url, json=data)
```

Sample run

```
❯ python3 test.py 5000 node1
 * Serving Flask app 'test'
 * Debug mode: off
timeouting in 14 seconds
Starting election in: node1
higher node array []
http://localhost:5000/announce
Coordinator is node1
Coordinator is : node1
**********End of election**********************
```

```
❯ python3 test2.py 5001 node2
 * Serving Flask app 'test2'
 * Debug mode: off
timeouting in 6 seconds
```

```
Starting election in: node2
higher node array [5000]
Coordinator is node1
```

Conclusion

In this way ”Lab6 :Implementation of Distributed Election Algorithm” was completed by implementation of Bully Algorithm. The Bully algorithm for leader election is a good way to ensure that leader-dependent distributed algorithms work well. The algorithm provides quick recovery in case leader nodes stop working, although the network usage is not very efficient.