Bully Leader Election Algorithm Implementation

CS403/534 - Distributed Systems
Programming Assignment (PA) #4, Fall 2020

January 4, 2021

Abstract

In this assignment, you will implement Bully Leader Election (BLE) algorithm[1]. The algorithm depends on a reliable multicasting mechanism which will be realized using ZMQ sockets' publish - subscribe mechanisms.

1 The BLE algorithm

In BLE algorithm, multiple nodes can start the leader election protocol simultaneously. When a node realizes that the leader is unavailable, it multicasts a `LEADER` message to all nodes with higher IDs. If the receiver is online, it responds with a `RESP` message. Therefore, if the initiator receives a response, it understands that a node with a higher ID is alive and it becomes passive listener.

When a node receives a `LEADER` message, it also multicasts another `LEADER` message to nodes with higher IDS. Similar to the initiator, if it receives a response, it understands that a node with higher ID is alive. Otherwise, it understands that it is the node with highest ID[2] and it broadcasts a `TERMINATE` message with its own ID. When other nodes receive the `TERMINATE` message, they learn the new message and they finalize the protocol.

Although the algorithm description is simple, the implementation depends on some detailed points explained in the next section.

2 Details

- **Input:** You are expected to write a single PYTHON file (named `bully.py`) containing all your implementation. As in PA4, your file should be callable from command line with the following arguments: *numProc*, *numAlive* and *numStarters*. *numProc* is the total number of nodes, *numAlive* is the number of nodes that are alive - online and *numStarters* are the number of nodes that will initiate the protocol. Obviously, $0 < numStarters \leq numAlive \leq numProc$ and you might assume that the client obeys this restriction while providing input.

[1] see Chapter 6 of your course book or slides
[2] For this algorithm, you can safely assume that node IDs are unique.

- In your main method, first you will pick random *numAlive* IDs from the range $[0, numProc)$ and then you will pick *numStarter* IDs from this set. You can use `random.sample` for this purpose.

- For each alive node, you need to spawn a new process from the main (executing the `leader` method). This process will implement the algorithm. However, single thread per process is not enough for implementing BLE. For each node, there should be a separate thread listening for concurrent messages from other nodes and responding to them while the main thread is multicasting `LEADER` messages and waiting for the responses. When a new process is spawned, it should print `PROCESS STARTS` with its Operating System process ID, node ID (input to the leader) and whether it is a starter or not (also an input to the leader)[3]. Also, when the listener thread (executing `responder` method) starts running it should print `RESPONDER STARTS` with node ID.

- For each process, the main thread (executing `leader` method) and the listener thread (executing `respond` method) can communicate using shared variables. If you detect any critical sections, you might use the `threading.lock` module.

- For interprocess communication (multicasting and pairwise messaging), you have to use ZMQ sockets with PUB-SUB options. Basically, each process uses the local host as the TCP address and for each process you need to assign a different port to bind i.e. $5550 + nodeID$. When publishing, a node can bind to its own port and when subscribing it should connect to all ports of all processes.

- **The `leader` method**: It is the main method of each process. It should first create the listener thread. Then, it waits until either it needs to multicast a `LEADER` message (which is the case if it is a starter node or the listener thread receives a `LEADER` message) or the listener receives a `TERMINATE` message. If it needs to multicast a `LEADER` message, this method must wait for a response from nodes with higher IDs. If no node responds, this method broadcasts a `TERMINATE` message with its own node ID. Since this method runs in the main thread, it should wait for the listener thread to terminate.

- **The `responder` method**: This is the method run by the listener thread. Basically, it connects to the ports of all alive processes and subscribes to `LEADER` and `TERMINATE` messages. When a message comes it processes the message. If it is a `TERMINATE` message, it notifies the main thread and terminates. Otherwise, if the sender has a lower node ID, it responds to the sender and notifies the main thread for multicasting a `LEADER` message if it has not done so yet.

- For both `leader` and `responder` methods, you are required to print all messages sent by these methods. These prints will be evaluated for grading. Of course, you need to print the sender processes' node IDs as well.

[3]See example runs below

- A common problem with ZMQ sockets that you also faced in PA3 is that if a message is sent before the intended recipient is connected, this message might be lost. To prevent this, you might develop a protocol to ensure connection of interested processes or you might let the process sleep for a small amount of time before sending a message so that the other processes could find time to connect.
- When the `leader` method multicasts a `LEADER` message, it starts waiting for a response and if it has the highest ID, it never gets the response. Since the receive methods of ZMQ are blocking by default, directly calling them might block a thread indefinitely. To prevent it, you can configure a timeout for receive method or you might utilize `zmq.poller` module[4]. In both cases, you need to arrange the timeout value carefully so that you need to give enough time to other processes for sending a response.

3 Example Runs

In this section, two example runs are provided for your convenience.

For the first example, $numProc = 10$, $numAlive = 4$ and $numStarter = 2$:

```
Alives:
[8, 9, 0, 3]
Starters:
[9, 3]
PROCESS STARTS: 42004 8 False
RESPONDER STARTS: 8
PROCESS STARTS: 49092 9 True
RESPONDER STARTS: 9
PROCESS STARTS: 42836 3 True
RESPONDER STARTS: 3
PROCESS STARTS: 40496 0 False
RESPONDER STARTS: 0
PROCESS MULTICASTS LEADER MSG: 9
PROCESS MULTICASTS LEADER MSG: 3
RESPONDER RESPONDS 9 3
PROCESS MULTICASTS LEADER MSG: 0
RESPONDER RESPONDS 8 3
PROCESS MULTICASTS LEADER MSG: 8
RESPONDER RESPONDS 9 0
RESPONDER RESPONDS 3 0
RESPONDER RESPONDS 8 0
RESPONDER RESPONDS 9 8
PROCESS BROADCASTS TERMINATE MSG: 9
```

For the second example, $numProc = 10$, $numAlive = 5$ and $numStarter = 3$:

```
Alives:
[2, 4, 1, 9, 0]
Starters:
```

[4] see `https://dev.to/dansyuqri/pub-sub-with-pyzmq-part-2-2f63` for more detail.

```
[9, 4, 2]
PROCESS STARTS: 22348 2 True
RESPONDER STARTS: 2
PROCESS STARTS: 42824 4 True
RESPONDER STARTS: 4
PROCESS STARTS: 40668 1 False
RESPONDER STARTS: 1
PROCESS STARTS: 47400 0 False
PROCESS STARTS: 48448 9 True
RESPONDER STARTS: 0
RESPONDER STARTS: 9
PROCESS MULTICASTS LEADER MSG: 2
PROCESS MULTICASTS LEADER MSG: 4
PROCESS MULTICASTS LEADER MSG: 9
RESPONDER RESPONDS 9 2
RESPONDER RESPONDS 4 2
PROCESS MULTICASTS LEADER MSG: 1
PROCESS MULTICASTS LEADER MSG: 0
RESPONDER RESPONDS 9 4
RESPONDER RESPONDS 2 1
RESPONDER RESPONDS 1 0
RESPONDER RESPONDS 4 1
RESPONDER RESPONDS 9 0
RESPONDER RESPONDS 4 0
RESPONDER RESPONDS 2 0
RESPONDER RESPONDS 9 1
PROCESS BROADCASTS TERMINATE MSG: 9
```

4 Submission Guidelines

PA4 must be implemented in Python using ZMQ sockets. You need to submit the following Python file:

- `bully.py`: A Python script that can be called from command line with arguments as specified in Section 2.

For this PA, you are allowed to work in groups of two. If you prefer, you can submit individually. Required files explained above should be put in a single zip file named as `CS_403-534_PA03_`*yourSUname*`.zip` and submitted to PA4 under assignments in SUCourse+. For groups, you can use name of both members or a single member but you need to put a text file in the zip with the names of group members for the second case. Submission will be open until January 11, 2021 23:50 Turkish time.

5 Grading Criteria

- Input Processing (10 pts): You process input parameters correctly. You generate a valid set of alive node IDs and a valid set of starters.

- Starting (20 pts): You create a process for each alive node that runs two threads: main and listener.
- Communication (20 pts): Multicast can successfully performed. Intended nodes can receive the messages and messages are not lost. If you use something other than ZMQ with PUB-SUB options, you will lose credits.
- Correct Result (30 pts): There is only one terminate message sent by the node with the maximum ID among the alive nodes.
- Termination (20 pts): All processes and threads eventually terminate. No thread/process should be left in a blocked state.