UiT INF-3200 Distributed Systems - Project 2 Fall 2015

Enrico Tedeschi
ete011@post.uit.no

Mike Murphy
mmu019@post.uit.no

I. Introduction

Our task was to implement leader election on top of a peer-to-peer network.

All the peers in the network must agree on who the leader is and only one peer can be leader at a time. Provides support for peers joining and leaving the network.

A. Requirements

- Support at least 10 nodes in a p2p network structure of your own choice. No centralized architectures allowed; i.e. all processes should behave similarly.
- Support graceful shutdown of nodes. On receiving a signal to shut down(SIGTERM), a node should leave the network.
- Support adding nodes on demand. Adding a new process allows you to grow the system as the demand increases.
- Leader election. There should at all times be a single leader. A pertinent Q: What happens if the leader leaves the network?
- A GET request to any node for the url "/getCurrentLeader" should return the ip and port of the current leader. The response body must be formatted as a single ip:port (e.g. "127.0.0.1:1234") entry.
- A GET request to any node for the url "/getNodes" should return a list of ip and port pairs of all nodes connected to the recipent node. The response body must be formatted as a list of ip:port (e.g. "127.0.0.1:1234") entries with newline separating each ip:port pair.
- Measure the time it takes to elect a leader when the number of nodes changes.

II. Technical Background

To solve the problem some technical background are required. First of all, a good knowledge about programming and some basic concept about distributed systems is necessary. Is good to know and to study then, some of the possible election algorithms which could be used. We took into consideration the *Bully algorithm* and the *Ring algorithm election*.

A. Bully algorithm

When a new leader is needed a process P send an election message to all the nodes with an higher ID number. If there are no answers, P wins the election. If P gets an answer then it terminates his job and the election continues with the node with the higher value just called. In the Fig 1 the node 3 starts the election because the previous leader 6 crashed. The new leader will be the node 5.

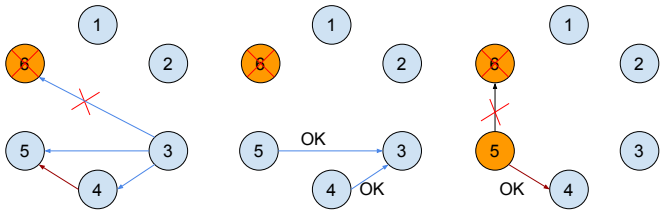

Fig. 1: Example of how the bully algorithm works, the leader changes from 6 to 5

B. Ring algorithm election

Every node needs to know only about his successor. When any process figures out that there is no leader anymore, it generates an election message. The message flows through the ring network and every node adds its ID in it. If a node finds its ID in the message it means that that node started the election, so it will be the new leader and a new message will be sent to announce the new coordinator. In the Fig 2 the node 5 realize that the coordinator is down. It start a new election and at the end it gets the leader role. Then it sends an ok message which tells to the other nodes who is the new coordinator.

A good knowledge of the problems that the scale of nodes could cause is also required, that could be related to security or data consistency. Even if the solution

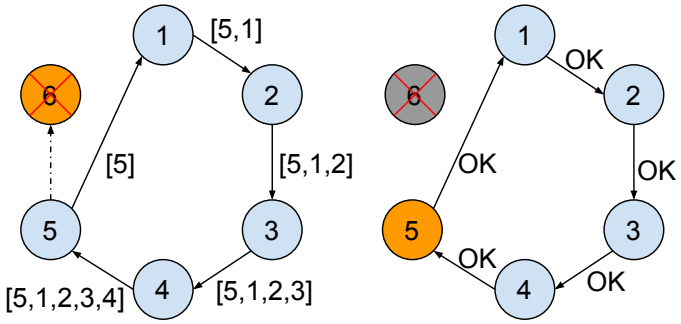

Fig. 2: Example of how the ring algorithm election works, the leader changes from node 6 to node 5, the election is started from the node 5

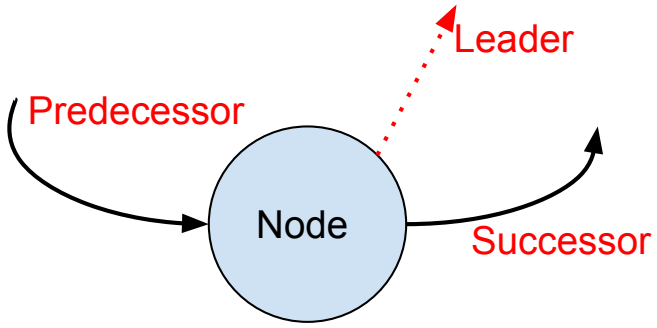

Fig. 3: Design and information stored in a single node

doesn't include the security and data consistency, it should be taken into consideration a good policy of adding and removing a node such as the way of setting the successor and predecessor for every join-leave.

III. Design

Since the precode was taken from the assignment 1, where a ring network was implemented, for simplicity we decided to keep this network topology also for the assignment 2. The ring algorithm election was chosen as a leader selection method; that because in our previously network topology each nodes knows about its successor, which is also the way that the ring algorithm election works. To implement the Bully algorithm we would have added further information for each node, such as a pointer to all the nodes with an higher ID than the current one.
In our implementation each node has the characteristic showed in Fig 3 so it knows only about the previous node, the next one and who is the leader.
The predecessor was implemented to allow an ordered join in the network.

The network was designed, as the Fig 4 shows, to implement the leader election in a ring network topology and it includes also a join and a graceful shut-down of the node. The detection of crashes has not been implemented in this version because not required. In the Fig 4 the leader is the *N_L* node and when the leader leaves the network a new election must be raised. Furthermore, an external node *N+1* can join the network in the ordered position according to its ID.

IV. Implementation

As already explained in Chap III, a ring network using a ring algorithm election with join and leaving features

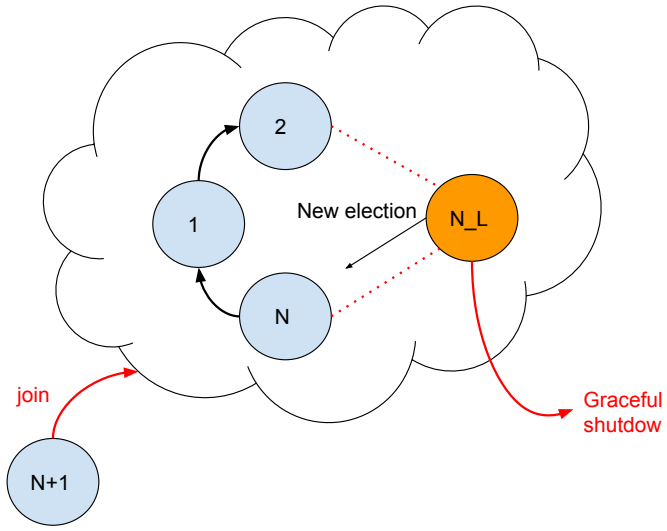

Fig. 4: Design of the implemented network

was implemented.

The python code is made by two main classes, one in which the core of the node is stored and another which contain the *HTML* parser to manage the communication between nodes. The core of the node allows to manage the following messages:

- Join
- JoinAccepted
- NewPredecessor
- Election
- ElectionResult
- GetNeighbors
- GetLeader
- NewSuccessor
- ShutDown

A. Languages and Code

Our solution was implemented in a mix of Python and Bash script, Python for the actual node implementation, and a Bash script to communicate through the network created, adding and removing nodes.

We started with skeleton code by our first assignment for what concerns the node and the script code. The code was rearranged by removing the front-end node and by adding some properties to the nodes such as the predecessor node and information about the leader, which at the beginning, is the first node joining the network.

The code were tested before on local machine, using bash scripts and the given visual test code from Einar Holsb Jakobsen and Magnus Stenhaug and then on the uvrocks cluster.

B. Leader election

The leader election is implemented by using the **ring algorithm election** (see Fig 2). Every node knows about which one is the leader and a new election is raised only if the current leader is gracefully shut down (chap IV-D). A new election is invoked from the next-node of the previous coordinator. The election will go through all the network and each node will add its ID in a pile. When a node find its ID in the pile it will be the new leader and it will send an *OK* message to all the other nodes, containing the information about who is the new leader. The Fig 6 shows the raising of a new election, where *N_L* was the previous leader and its successor, *N+1*, starts a new election.
Since the network doesn't detect any crashes, but allows a graceful shut-down of the nodes, the problem that two nodes realise that the coordinator is missing is not taken into consideration.
The following pseudocode is called when an *Election* message type is raised and represent the core implementation of the election algorithm.

```
if no successor
    # Single node. You are already the leader
    return OK
if ID in message
    # You win. Create announce message
    announce = ElectionResult(node.next - OK)
    return OK -- with announce
else
    # add your name and forward.
    fwd = Election(add ID - node.next)
    return OK -- with fwd
```

C. Join

To implement the scaling of nodes, a *join* function is fundamental together with the *leave_network* one. A node can join in every part of the network. Each node has a personal ID obtained by **hashing its IP address**. To guarantee an ordered join in the network, when a node N_j asks to join to a node N_i , it checks that its ID is in between the ID of N_i and the ID of N_{i+1} .
It is clear then, that a node N_j is insert in the *i* position only if the following formula is respected:

$$N_i < N_j < N_{i+1}$$

where N_i is the ID number of the node which got the join request, N_{i+1} the ID of its successor and N_j the ID of the node who wants to join the network. Once that the right place is found, the properties of the new network are setted as the following:

$$successor(N_j) \leftarrow successor(N_i)$$
$$predecessor(N_j) \leftarrow N_i$$
$$leader(N_j) \leftarrow leader(N_i)$$
$$successor(N_i) \leftarrow N_j$$
$$predecessor(N_{i+1}) \leftarrow N_i$$

A graphic implementation of the join is represented in the Fig 5.

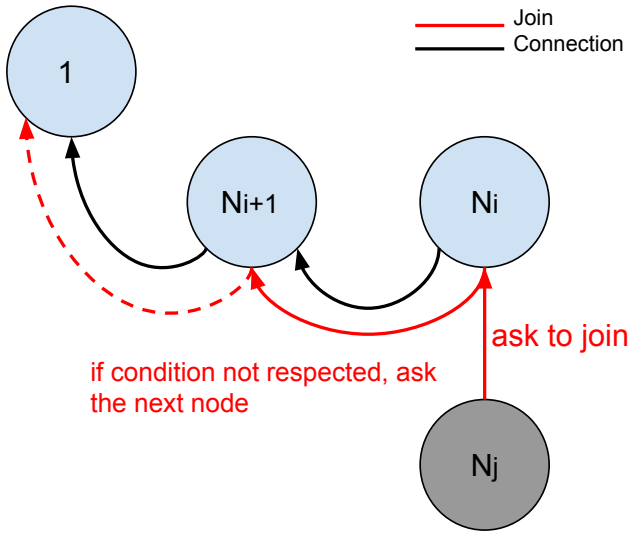

Fig. 5: Implementation of the join when a node N_j ask a to a node N_i to join.

D. Graceful shut-down

The other part involved for guarantee the scalability is the *node leaving* function. As said before, we don't provide fault tolerance and only a graceful shut-down

of the nodes is implemented. When a simple node is shut-down, before leaving the network, it gives the information about its predecessor and successor to its neighbours. If is the coordinator that leaves the network (like in Fig 6) then the successor of the leaving node raise an election message.
When a node N_i leaves the network the following equations are implemented:

$$successor(N_{i-1}) \leftarrow N_{i+1}$$
$$predecessor(N_{i+1}) \leftarrow N_{i-1}$$

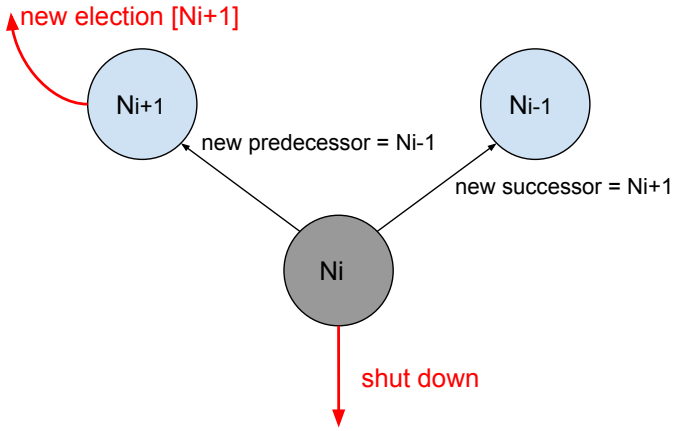

Fig. 6: Example of graceful shut-down on the node N_i, which is also the leader.

E. Node

F. Environment

Our code was written to run on the Rocks Cluster distribution[1], and makes some assumptions about that environment. We rely on the cluster's shared filesystem for distributing program code to servers. And we rely on easy SSH access between machines in the cluster to start and shutdown nodes.

V. Discussion

VI. Evaluation

For the evaluation node scaling has been considered. The function *storage_frontend* has been timed sending five hundred requests (GET/PUT) to the nodes network. The evaluation has been done considering the scale on the number of nodes and for each, 10 tests were taken into consideration.
The number of nodes and the average time of 10 computations is represented in the table in Fig 7.

In the Fig **??** instead the graphic of this scaling test is characterized.

Fig. 7: Nodes/Time scaling table

Nodes	Time
2	5.7923
4	6.4793
6	8.1309
10	13.0746
15	17.0469
20	19.3472
30	29.4729
40	35.2886

VII. Conclusion

Our DHT solution, with a simple ring structure, was able to store and retrieve data correctly, in time that increased linearly with the number of nodes ($O(n)$).

References

[1] Rocks clusters: About. http://www.rocksclusters.org/wordpress/?page_id=57.

[2] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking up data in P2P systems. *Commun. ACM*, 46(2):43–48, February 2003. http://doi.acm.org/10.1145/606272.606299.

[3] Frank Dabek, Emma Brunskill, M. Frans Kaashoek, David Karger, Robert Morris, Ion Stoica, and Hari Balakrishnan. Building peer-to-peer systems with chord, a distributed lookup service. https://pdos.csail.mit.edu/papers/chord:hotos01/hotos8.pdf. MIT Laboratory for Computer Science.

[4] Ronald Rivest. The MD5 message-digest algorithm. RFC 1321, RFC Editor, April 1992. http://dx.doi.org/10.17487/RFC1321.

[5] Sean Turner and Lily Chen. Updated security considerations for the MD5 message digest and the HMAC-MD5 algorithms. RFC 6151, RFC Editor, March 2011. http://dx.doi.org/10.17487/RFC6151.