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.

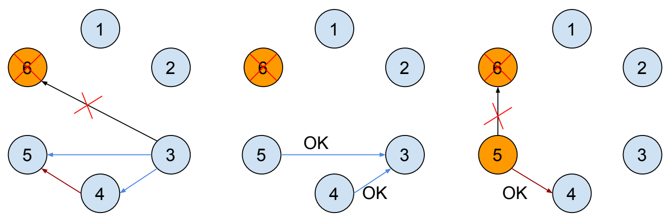

Fig. 1: Example of how the bully algorithm works, the leader changes from 6 to 5

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.

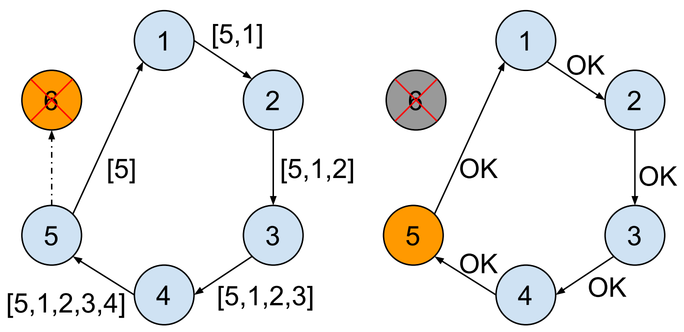

Fig. 2: Example of how the ring algorithm election works, the leader changes from node 6 to node 5, the election is started from the node 5

III. Design

IV. Implementation

A. Languages and Code

Our solution is 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. Network Protocol

C. Persistence

D. Frontend

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 3.

Fig. 3: 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

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

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.