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.

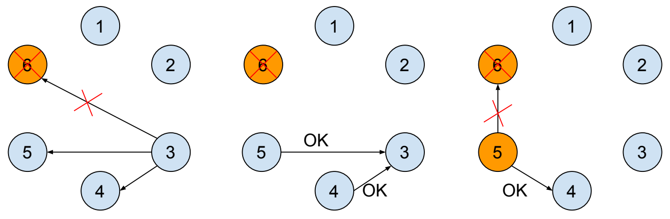

Fig. 1: Example of how the bully algorithm works, the leader changes from 6 to 5

B. Ring algorithm election

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

The backend nodes accept and retrieve data through a simple HTTP API. HTTP's PUT and GET operations are a natural fit for a key-value store, as their semantics specify storing and retrieving documents (value) at a given URL path (key). Jakobsen and Stenhaug chose this protocol for their starter front-end node and we reused it for the storage nodes.

C. Persistence

The purpose of this exercise was to investigate the challenges of distributed data storage, not storage itself. Therefore, there was no requirement to actually persist stored data between runs. So, for simplicity, we did not implement any kind of data persistence. Data is simply stored in memory and the store starts empty on each test run.

D. Frontend

As specified in the requirements, the *Frontend Node* (Fig **??**) contacts a random backend node for each request, and the backend nodes must cooperate to handle the request. This randomness enforces the distribution transparency of the data store. The system can store and retrieve keys as a whole, no matter which individual node is contacted.

E. Node

Each node (Fig **??**) has a simple workflow. It starts running and waits for a request, which could be GET or PUT from the frontend node or another backend node. For each request, the node hashes the key into the linear key space, and checks if it falls within the region assigned to that node. If so, the node handles the request by storing the value (for PUT) or returning a previously stored value (for GET). If the key does not fall in the node's assigned range, it forwards the request to the next node (Fig **??**).

The node's logic is split across two classes: *NodeCore* and *NodeHttpHandler*. *NodeCore* includes the core logic of deciding when to store a key and when to forward a request to the next node. *NodeHttpHandler* includes the logic of interpreting HTTP requests and formatting HTTP responses. This separation of logic makes it possible to verify the core algorithm with isolated unit tests, without having to set up HTTP servers.

For hashing, the node uses the MD5 algorithm to map the string key to a numeric value, then takes that large integer modulo the number of nodes to get a node number (Fig **??**). As a cryptographic algorithm, it is deterministic and it will evenly distribute the keys. MD5 is also fast[4], so it will not slow down lookups unnecessarily. And though it no longer considered suitable for security applications[5], it is not being used for security here. We merely need to decide which node should store a given key.

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

Our design decisions favored simplicity over performance. The simple ring structure was easy to implement, but requests may need to be forwarded along several nodes in the ring before finding the correct key. The number of hops is linear with the number of nodes, $O(n)$. Chord's finger-table optimization[3] finds nodes in $O(\log_2 n)$ hops. We had hoped to implement the same strategy in our project but we ran out of time.

The simple synchronous request forwarding strategy is also a potential bottleneck. It leaves a thread idle on each node involved in each request, waiting until the right key is found before returning it back through all of the nodes that were involved. We suspect that, as the number nodes grows or request frequency increases, this holding of resources will choke the system. The advantage of this approach is its simplicity. The request and response protocol is identical between client and front-end, front-end and storage node, and between storage nodes themselves. The storage nodes also use the same logic to handle requests whether the request comes from the front-end or another node in the chain.

Asynchronous message-passing would allow intermediate nodes in a search to free resources. The first node could block, and then send a message to the next node. If this message included a return address to the first node, then the other nodes in the ring could pass it along without leaving connections open or blocking. Finally, the node that has the key could send the value to original node. When that original node received its answer, it could unblock and send the response to the front-end. Such a strategy should be more efficient, but would require different message formats for requests and answers, and nodes would need logic to receive answer messages and match them to waiting threads to send responses. We suspect that the finger tables would give a much larger performance boost. Not only that, but with $O(\log_2 n)$ searching, only a few nodes would be involved in each search, and the overhead from waiting synchronously on just a few nodes might be acceptable.

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 2.

Fig. 2: 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.