Course Project:
Distributed Mutual Exclusion: Token Passing on a Tree

Distributed Systems 1, 2018 – 2019

In this project, you will use Akka to implement distributed mutual exclusion (DMX). DMX is employed in various systems to grant privileges safely to a single entity. The entity holding privilege is the only one that can enter a critical section (CS), i.e. the only one that is allowed to access a shared resource that needs to be protected against concurrency. Example applications for DMX are grid computing, medium access control, collision avoidance in wireless broadcasts, distributed databases, and management of replicated data.

A key metric used to evaluate DMX protocols is the number of messages required for a given requester to enter the CS. Lamport's algorithm[1] is a widely known permission-based DXM protocol that requires $3(N-1)$ messages. Lamport's nodes explicitly accord permission to enter CS; another possibility is token passing. In this case, the node holding the token is the only one that has access to the shared resource and mutual exclusion is automatically ensured. However, if nodes are arranged in a ring topology, the number of messages required is still $O(N)$. For systems composed of many nodes, linear message complexity may be unacceptable. Raymond's algorithm[2] is token-based, but connects nodes using a tree instead of a ring, achieving $O(\log N)$ message complexity. A node trying to enter a CS needs to ask for the token first. The basic idea of this protocol is to establish directed edges that always point towards the current token owner, so that requests for privilege reach destination in few hops along the tree. You are asked to implement mutual exclusion based on Raymond's algorithm. The operations you have to support are summarized in this document. You can refer to the original paper for a detailed description.

Protocol overview

Initialization. We assume that nodes in the system do not change over time, i.e., nodes do not join or leave. The established tree topology is fixed as well. First, you should create the tree topology. Each node must know its neighbors. Once all nodes have been created, the token needs to be injected. Choose an arbitrary node and make it the initial owner of the token. Next, the owner should inform all other participants of the token location by flooding along the tree. After initialization, all nodes should be able to tell which of their neighbors is closer to the token owner. Each node has a variable HOLDER that indicates the position of the token relative to the node itself and therefore defines directed edges. For any node, HOLDER is always one of its neighbors.

Requesting the token. Either a node holds the token, or it knows which of its neighbors sub-tree contains it. A token REQUEST message is sent towards the current owner and relayed by nodes along the path defined by HOLDER variables until it arrives to the token location. In practice, a chain of requests is formed between the node willing to enter the CS and the current token owner. Each node can be in charge of multiple token requests from different neighbors.

Granting the token. When the current token owner receives a REQUEST message, it queues that request until it does no longer require the token to execute the CS. Then, it replies with a PRIVILEGE message back to its neighbor that is the sender of the earliest REQUEST in the queue. Note that holding the token does not imply being inside the CS. When no node wishes to enter the critical section, the last node to use the privilege continues to hold it. The node receiving the PRIVILEGE message becomes the new unique owner of the token, and tree edge directions are updated accordingly by properly setting the HOLDER variable. If the new owner requested the token on behalf of another node, it forwards the PRIVILEGE message.

Node failure and recovery

Nodes may fail and restart, losing HOLDER information. That is, they may become incapable of locating the token. If a node undergoes failure while holding the token, the token is lost. Assume that the node becomes available again, retaining the references to its neighbor. To recover from failure, the node collects information from neighbors and infers the value that its HOLDER variable should be set to. The queue of pending requests of

[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system", Communications of the ACM, vol. 21, no. 7, pp. 558-565, 1978.

[2] K. Raymond, "A tree-based algorithm for distributed mutual exclusion", ACM Transactions on Computer Systems, vol. 7, no. 1, pp. 61-77, 1989.

the recovering node must be reconstructed, but the original order of elements in the queue should be disregarded.

Requirements and assumptions

- Message complexity: the system requires $O(\log N)$ messages to enter the CS for a given node.
- Concurrent REQUEST messages: the system supports any number of token requests coming from any set of nodes; these requests should not be lost.
- No duplicated REQUEST messages: if a node is already waiting a PRIVILEGE message, it should not send additional REQUEST ones.
- FIFO privilege: each node keeps a queue with pending token requests to satisfy them in order of arrival; you can refer to the pseudocode in the original paper of Raymond for hints about this aspect (REQUEST_Q data structure).
- Entering and executing the CS: to easily test your implementation, you should be able to directly ask a node to enter the CS; first, the node adds itself to the local request queue. If the node is not holding the token and has not requested it yet, it sends a REQUEST message. Execution of the CS is simulated; the node remains inside it for a configurable duration (while still processing incoming messages), and logs both when it enters and when it exits.
- No failures within CS: assume a node never fails while inside CS. It may fail at any other time.
- Single failure: assume nodes cannot fail while another one is unavailable or recovering. A node does not fail during the recovery process.
- No global variables: your implementation cannot rely on global variables accessible by different nodes; of course, you can use configuration variables defined at compile time.

Grading and the levels of complexity

You are responsible to show that your project works. The project will be evaluated for its technical content (algorithm correctness). *Do not* spend time implementing unrequested features — focus on doing the core functionality, and doing it well.

A correct implementation of the whole requested functionality is worth 6 points. It is possible to submit a project with a reduced level of complexity, supporting a single token request at a time (the next one happens after the requesting node exits CS) and without the simulation of node failure and recovery. A correct implementation of this reduced function set is worth 3 points.

You are expected to implement this project with exactly one other student, however the marks will be individual, based on your understanding of the program and the concepts used.

Presenting the project

- You MUST contact through e-mail the instructor (gianpietro.picco@unitn.it) AND the teaching assistants (davide.vecchia@unitn.it, timofei.istomin@unitn.it), well in advance, i.e. a couple of weeks before the presentation.
- You can present the project at any time, also outside of exam sessions. In the latter case, the mark will be "frozen" until you can enroll in the next exam session.
- The code must be properly formatted otherwise it will not be accepted (e.g. follow style guidelines).
- Provide a short document (1-3 pages) in English explaining the main architectural choices.
- Both the code and documentation must be submitted in electronic format via email at least one day before the meeting. The documentation must be a single self-contained pdf/txt. All code must be sent in a single tarball consisting of a single folder (named after your surnames) containing all your source files. For instance, if your surnames are Rossi and Russo, put your source files and the documentation in a directory called `RossiRusso`, compress it with "`tar -czvf RossiRusso.tgz RossiRusso`" and submit the resulting `RossiRusso.tgz`.
- The project is demonstrated in front of the instructor and/or assistant.

Plagiarism is not tolerated. Do not hesitate to ask questions if you run into troubles with your implementation. We are here to help.