UNIVERSITY OF TARTU
Institute of Computer Science
1632

Distributed Systems (Spring 2022)

Mini-project 1: Mutual exclusion and broadcast

Practical information:
Due date: Monday, April 4, midnight (FIRM DEADLINE) – Zero points after deadline
- This task can be performed individually or in a team of max 2 persons.
- Total amount of pts that that can be collected = 100 + Bonus pts = 125 pts. **Bonus pts are optional**
- Programming languages that can be used: **Java/Python**

Submit your solution to zhigang.yin@ut.ee, mohan.liyanage@ut.ee, (CC) huber.flores@ut.ee

Instructions: Please respond to the following questions. **Be clear and concise in your answers.** Provide enough explanation to support your arguments. Ambiguous answers to fill up space are considered wrong and no points are granted. For questions that involve implementation, be sure to include everything needed to execute your program **(Re-submissions are not allowed)**.

1. **Mutual exclusion in distributed systems**

Computer nodes in a distributed system rely on mutual exclusion algorithms to access (centralized) resources in an exclusive manner before using them. There is a variety of mutual exclusion algorithms available to handle this issue. For instance, token ring and permission-based algorithms are some of the most common.

Distributed algorithm: Ricart & Agrawala algorithm can be utilized to deal with mutual exclusion of distributed nodes. The algorithm relies on exchange of messages using broadcast communication to regulate the access of each node to the critical section (CS) of the system. In this task, the algorithm must be implemented.

Concept: The implemented algorithm requires to follow the description based on slide 35 of "Lecture 6 – Election algorithms". This is to avoid implementing slightly different versions of the same algorithm. The basic idea of the algorithm is as follows, a group of processes sharing a critical session (resource), need to exchange messages with (Lamport-like) timestamps to coordinate exclusive access to a resource (see Figure below). The one with the lowest timestamp (process waiting to enter the CS) is the one that is granted access to the CS. A process can have three different states when coordinating mutual exclusion, HELD (The process has locked the CS), WANTED (The process wants access to the CS), and DO-NOT-WANT (The process is not interested to enter the CS – but can switch to WANTED access later on)

Requirements: The implementation must contain the following functionality. The program is required to provide a command line interface, and must implement the following functionality.

###Starting the program
$ **Your_RA_program.sh** N (75 pts)
- The program receives N as a parameter for starting its execution. Here, N is the number of processes (each process is a thread) that are created concurrently. N cannot be zero (N>0). All of these processes will be competing for accessing the (fixed) critical section (CS)

- Each process can have three states, HELD, WANTED and DO-NOT-WANT. Each process changes between different states, and the switch of state is ruled by a fixed time-out. The initial state of a process is DO-NOT-WANT. After a time-out, each process changes to WANTED and starts the requesting for access to the critical. The time-out is a random period of time selected in the interval [5, t] in seconds, where 5 is lower bound and **t** is the upper bound. By default, as the program starts, both are fixed at 5. Once a process is granted access to the critical section, then it accesses the CS for a period of time (state is now HELD), and then it releases the CS automatically (after a period of time fixed by the CS), going back to its initial DO-NOT-WANT state.

After the program has started, it has also to support the following commands via input terminal:

$ List: This command lists all the nodes and its states. For instance, (15 pts)

$ *List*
P1, DO-NOT-WANT
P2, DO-NOT-WANT
P3, DO-NOT-WANT
P4, DO-NOT-WANT
P5, DO-NOT-WANT
P6, DO-NOT-WANT

After t seconds, list command again

$ *List*
P1, HELD
P2, DO-NOT-WANT
P3, WANTED
P4, DO-NOT-WANT
P5, WANTED
P6, DO-NOT-WANT

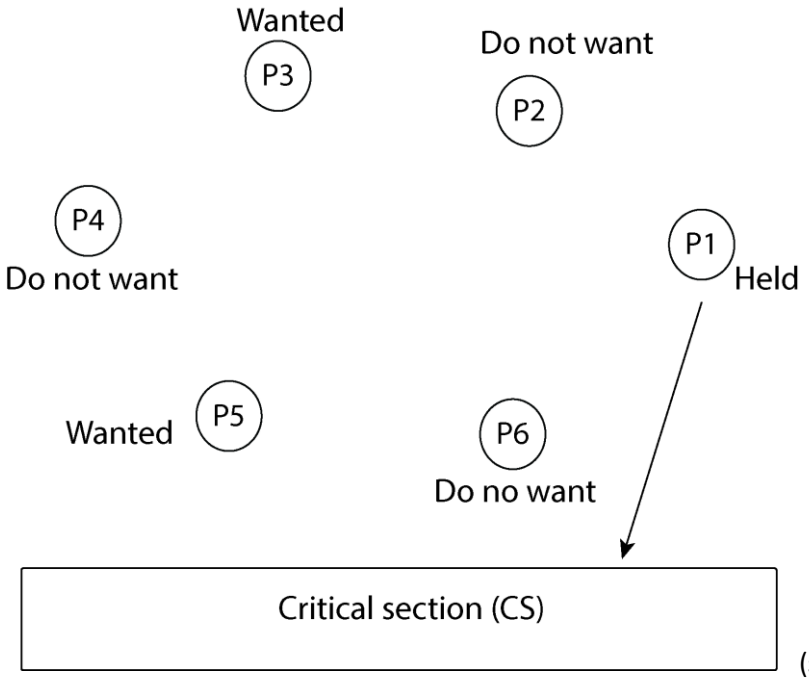
$ *time-cs* **t** (5 pts)

This command sets the time to the critical section. It assigns a time-out for possessing the critical section and the time-out is selected randomly from the interval (10, t). By default, each process can have the critical section for 10 second. For instance, $ time-cs 20, sets the interval for time-out as [10, 20] – in seconds.

$ *time-p* **t** (5 pts)

This command sets the time-out interval for all processes [5, t], meaning that each process takes its time-out randomly from the interval. This time is used by each process to move between states. For instance, a process changes from DO-NOT-WANT to WANTED after a time-out, e.g., after 5 seconds. Notice here that the process cannot go back to DO-NOT-WANT, and can only proceed to HELD once is authorized by

all the nodes to do so. After the process has going through the steps of accessing and releasing the CS, then it goes back to the DO-NOT-WANT, where the time-out can be once again trigger to request access to the CS.



(see slides 37-43 of Lecture 6)

Constraints: The algorithm should be implemented using explicit communication primitives (RPC).

Deliverables for your Ricart-Agrawala algorithm:

1- A short 1 min video showing your program in execution (showing all the commands)
2- Source code and everything needed for executing your program (binaries and input file) – in GitHub. Please also commit regularly, such that it is possible to track the contributions of each team member (if more than one).
3- We will evaluate your work with multiple file configurations. Thus, do not hard code your solution just to work with the input file described here.

2. **(25 pts) BONUS POINTS (OPTIONAL):** Install Apache Zookeeper and write a detailed tutorial of the configuration. The tutorial also needs to be accompanied by a small video showing Apache Zookeeper working. Apache ZooKeeper is an open-source server for highly reliable distributed coordination of cloud applications. ZooKeeper's architecture supports high availability through redundant services. The clients can thus ask another ZooKeeper leader if the first fails to answer.