UNIVERSITY of STIRLING

CSCU9V5: Concurrent & Distributed Systems

Assignment 2 -- Distributed Systems

This assignment covers material presented during the lectures on distributed systems and builds upon the work in the practicals on *distributed systems*. The deadline for this assignment is

Sunday, 29th November 2020 at 23:59

After submission you will give a mandatory demo to demonstrate and discuss your solution to the given problem in the light of the content of the module on distributed systems (weeks 8-11). A schedule and organisation of the demos will be communicated later on. Demos will be scheduled in the CSCU9V5 available slots of week 12 (starting 30th November).

Your submission should include:

1. a single pdf document, named **<your student number>-V5-report-2020.pdf** containing:
   i. a **short report** (roughly **four pages** plus a cover sheet with your student number) discussing the problem, any assumptions you made, and a description of your solution;
   ii. the **code listings** of your program. Please use appropriate report headings.

2. a zip file of your **source code** (for reference purposes only) named
   <your student number>-V5-code-2020.zip
   Please add your student number to each java file you will submit.

The short report should include appropriate diagrams of the design and screen shots of your application. Describe the various classes, their relationships, and outline the most important class methods. The report should explain how complete your solution is, and if applicable, any special cases when your program is not functioning correctly. It is important that your program code is properly commented, formatted and neatly structured for readability.

Considering the submission **and the demo**, you will receive marks for:

- the efficacy of the code (basic solution) 45%
- advanced features (**chose one!**) 25%
- the report 25%
- code comments and structure 5%

In the demo you may be asked to present and discuss your solution. The mark will consider presented code and the discussion.

Assignment Problem

Basic Problem

This assignment extends the theme of the labs on distributed systems. It builds on top of

- the first lab on distributed systems and the slides on sockets. It will adopt the same kind of client-server architecture explained in the slides and tested in the lab. You can find useful information in that teaching material.
- the Distributed Mutual Exclusion (DME) algorithm seen in lab 3, based on a token ring with a class implementing a *ring node,* and a *start manager* class to inject a token to start off the system. A variant of that algorithm will be developed here.

In this assignment you will develop *a traffic light centralised controller* that works like a *centralised DME.* The system will consists of two traffic lights and a coordinator. The rule is that only one traffic light at the time can expose the green light and let the cars that it controls go. In a word, the green light of the two traffic lights must be mutually exclusive.

In order to guarantee the correct functioning of the traffic lights, a *coordinator* passes a *unique token* to each one of the two traffic lights, exclusively. Traffic lights may require the token when needed, in the client-server style seen in lectures and labs. When granted the unique token, a traffic light will be allowed to switch the green light on, keep it on for a while, and then switch the green light off (no car is allowed to pass), and return the token to the coordinator. If a traffic light does not have the token, it cannot switch the green light on. In a sense, switching the green light on is a critical section that can be performed only by one traffic light at the time.

The coordinator program and the traffic light programs run on different JVMs. In the real world, these JVMs may run on different hosts, but for this assignment you can run them all on the same host, i.e. your computer. We call the traffic light programs *nodes.* The two nodes are connected to the coordinator. Sockets are used for communications and synchronisation, as seen in the labs.

You are issued with a skeleton solution, with incomplete programs for the coordinator and the nodes, as a starting point. It is strongly recommended that you study the details of the problem and the general functioning of the provided programs, before coding your solution. Useful suggestions are in the provided files under comments marked by >>>.

The skeleton solution proposed is general and supports nodes running on different hosts (but you will use only one computer). Therefore, each node has associated a specific *host,* i.e. an *ip address*, and a *port.* The *port* must be instantiated at launch time, e.g. as a parameter passed to the node. Ports 7000 and 7001 are used by the coordinator (assuming they are free on your computer!).

Each *node* (not the coordinator!), which resembles the client in the client-server architecture seen in labs, will perform the following loop:

1. *request the token from the coordinator*. This is done by sending the node's ip and port to the coordinator (the coordinator listens on port 7000 at a known ip, i.e. localhost).

2. *wait until the token is granted by the coordinator.* Granting the token can be implemented by a simple synchronisation by the coordinator on the node's ip:port, analogously to what done for the socket-based token ring that you developed in the labs.
3. *execute the critical section*, simulated by printing the message "Green light ON", sleeping for a few seconds (3-5 say), and printing the message "Green light OFF". These two messages clearly mark the beginning and end of the critical section, as done in the case of the DME based on a token ring, seen in the lab.
4. *return the token to the coordinator.* This can also be done by means of a synchronisation message (the coordinator listens on port 7001), as done in the case of the DME based on a token ring, seen in the lab.

Important: please run the two nodes and the coordinator on three different windows, open more console windows (e.g. in eclipse) or use three terminals/CMD in three different windows. It must be evident that only one node at the time is executing the critical section. Adapt, if needed, (random) timings to facilitate the understanding of programs' execution, and to test your system.

The *coordinator* consists of two concurrent tasks that share a *buffer* data structure (class *C_buffer*). It resembles the server seen in the lectures and labs:

- a *receiver* that continuously listens for requests from nodes. Requests consist of *ip* and *port* sent through a socket on port 7000. On connection, the *receiver* will spawn a thread of the class *C_connection_r*, as seen in lectures and labs. Such a thread receives *ip* and *port*, and stores them in the buffer using a *request* data structure, which is defined in the provided code.
- a *mutex* thread that continuously fetches requests from the buffer, if any (!), in a FIFO order. For each request, the *mutex* grants the token to the requesting node, by a simple synchronisation to the node's indicated port. Then, it waits for the token to be returned by means of a synchronisation on port 7001.

All sockets/servers must be suitably closed. All exceptions' *catches* must print appropriate messages, declaring occurred exceptions, if any! All components must print suitable messages showing their activities as suggested, e.g. start and stop of a critical section for nodes, and granting and receiving the token back for the coordinator.

Add some suitable random sleeping times and keep them varied: the order of requesting and granting the token to nodes should not be fixed. Please, test the case of one "very quick" traffic light that issues requests very frequently and one "very slow" traffic light, which issues request with a lower frequency. Verify that the fast one can switch the green light on more frequently than the slow one.

All relevant primitives, e.g. a synchronisation communication, have been seen in labs.

Develop *a socket-based centralised DME* as described above, based on the skeleton code provided.

Advanced Features

You can enhance the basic solution by implementing one of the following two advanced features:

1. Traffic lights are a bit less clever: nodes do not ask for the token (we can imagine that they usually do so depending on the number of waiting cars) but the coordinator grants the token to each of them alternatively. The coordinator waits for two nodes to register and the two nodes initially have to register with the coordinator by communicating their ip and port - they can use the same mechanism as in the previous case, but the queue of pending requests is not needed anymore.
2. Describe in the report, in an extra and properly marked section titled **Advanced feature 2**, what would happen if the coordinator crashes. Be as precise as possible and consider **all the possible relevant cases**, depending on when the coordinator crashes. You may want to report snapshots of the execution of your system to illustrate your findings (you can add extra pages for these snapshots).

NOTE: basic solution and advanced feature 1 must be developed in separate files in different, clearly named directories. Advanced feature 2 must be presented in an extra section of the report.
Please, choose and address ONE of the two ADVANCED FEATURES above, ONLY.

Plagiarism

Work that is submitted for assessment **must be your own work**. All students should note that the University has a formal policy on plagiarism, which can be found at
http://www.stir.ac.uk/academicpolicy/handbook/assessmentincludingacademicmisconduct/#q-8.

Plagiarism means presenting the work of others as though it were your own. The University takes a very serious view of plagiarism, and the penalties can be severe.

We check submissions carefully for evidence of plagiarism, and pursue those cases we find. Several students received penalties on their work for plagiarism in past years. Penalties range from a reduced mark, through to no mark for the module, to being required to withdraw from studies.

Submission on CANVAS

Please ensure you submit your assignment on CANVAS before **17:00 on 29th November 2020**, according to the submission guidelines reported above.

Late submission

If you cannot meet the assignment hand-in deadline and have good cause, please contact the module coordinator by e-mail (Dr. Andrea Bracciali) before the deadline to explain your situation, and ask for an extension through the Extension Request service on Canvas. Coursework will be accepted up to seven calendar days after the hand-in deadline (or expiry of any agreed extension), but the grade will be lowered by 3 marks per day or part thereof. After seven days the work will be deemed a non-submission.