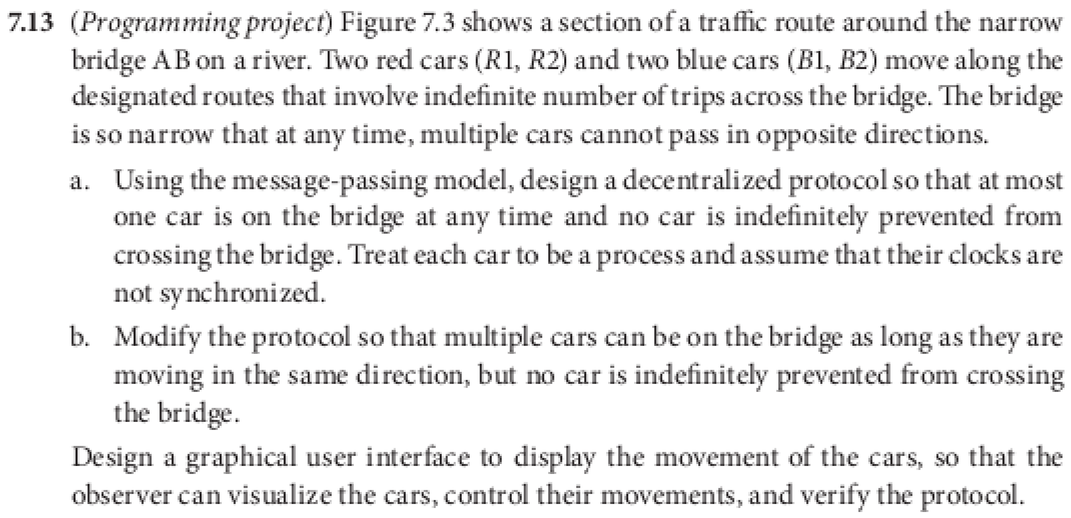
Car2Car Simulator, using Lamport's Mutual Exclusion Algorithm for Distributed Systems

By Shaomin (Samuel) Zhang, Mar 2015

Brief

This is a programming project in my Distributed Systems course. The purpose of this project is to better understand the algorithms for Distributed Mutual Exclusion. Lamport's algorithm for mutual exclusion is chosen for my program. This program consists of two parts. The "Car" runs at backend and its modules include several socket message channels, Lamport's logical clock, Lamport's Mutual Exclusion algorithm and its modified version for the second part of this question, and car's basic functionality movement. The other part is a GUI application based on a 3rdparty GUI system called FTK, which is an open source cross-platform GUI tool that I was once familiar. These two parts of programs are actually independent but highly integrated together and can run on Ubuntu Linux distribution.

Problem



7.13 (*Programming project*) Figure 7.3 shows a section of a traffic route around the narrow bridge AB on a river. Two red cars (*R*1, *R*2) and two blue cars (*B*1, *B*2) move along the designated routes that involve indefinite number of trips across the bridge. The bridge is so narrow that at any time, multiple cars cannot pass in opposite directions.

a. Using the message-passing model, design a decentralized protocol so that at most one car is on the bridge at any time and no car is indefinitely prevented from crossing the bridge. Treat each car to be a process and assume that their clocks are not synchronized.

b. Modify the protocol so that multiple cars can be on the bridge as long as they are moving in the same direction, but no car is indefinitely prevented from crossing the bridge.

Design a graphical user interface to display the movement of the cars, so that the observer can visualize the cars, control their movements, and verify the protocol.

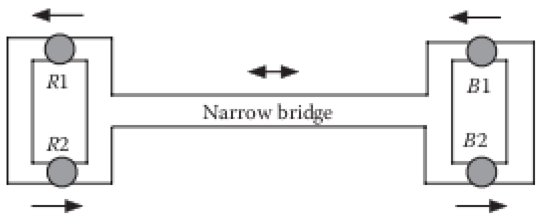


FIGURE 7.3 Cars crossing a narrow bridge on a river.

Design

Lamport's Algorithm for Mutual Exclusion

Ghosh, Sukumar (2014-07-14). Distributed Systems: An Algorithmic Approach, Second Edition (Chapman & Hall/CRC Computer and Information Science Series) (Page 130-131). Chapman and Hall/CRC. Kindle Edition.

7.2.1 Lamport's Solution

The first published solution to this problem is due to Lamport. It works on a completely connected network and assumes that interprocess communication channels are FIFO.

Each process maintains its own private request-queue Q. The algorithm is described by the following five rules:

LA1: To request entry into its CS, a process sends a time-stamped request to every other process in the system and also enters the request in its local Q.

LA2: When a process receives a request, it places it in its Q. If the process is not in its CS, then it sends a time-stamped ack to the sender. Otherwise, *it defers the sending of the ack until its exit from the CS*.

LA3: A process enters its CS, when (1) its request is ordered ahead of all other requests (i.e., the time stamp of its own request is less than the time stamps of all other requests)
in its local Q and (2) *it has received the acks from every other process* in response to its current request.
LA4: To exit from the CS, a process (1) deletes the request from its local queue and (2) sends a time-stamped release message to all the other processes.
LA5: When a process receives a release message, it removes the corresponding request from its local queue.

Design: Essential Factors for the Car

1. A car owns these properties:
    a. Lamport Logical Clock, which is to generate and maintain stable timestamps.
    b. State Machine with 3 states: *Out-of-CS, Waiting-for-Entering-CS and In-CS.*
    c. Network connections with every other car, which is able to both receive and send messages based on TCP. The message at least contains 3 parameters: sender's ID, message (*Request, Ack or Release*), timestamp.
    d. At least 3 queue-like buffers to maintain: requests from others, acks feedback from others for my own request and *the requests that I received in CS and should be acked after I leave CS*.
2. A car has other properties not related with critical section issue:
    a. Basic items from the requirements, like color, number, ID, (x, y)-coordinates, a timer-driven movement within a given map.
    b. Heartbeat report to a GUI observer, an independent GUI process who is only responsible for showing but not work as an coordinator or any kind of message mediator, so that the whole scenario can be seen by us.

Supplementary Design for Part (B)

Slightly we need to change the answer to such a question: "should the car in CS give an ACK to other requests?" For the previous version, the answer of course is absolutely No (Fig-A). For part-B of this question, now we just change it to "If the car who sends request is moving in the same direction with me, I, the car in CS, will give it an ACK", anything else keeps the same (Fig-B).

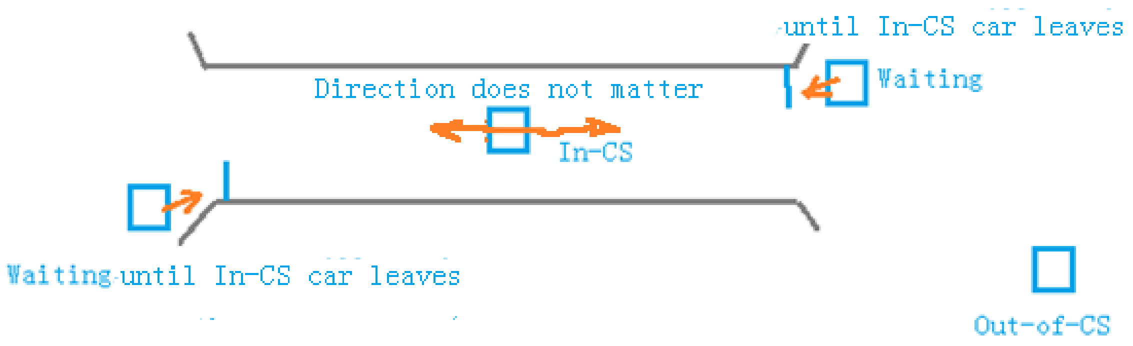


Fig-A At most one car is on the bridge

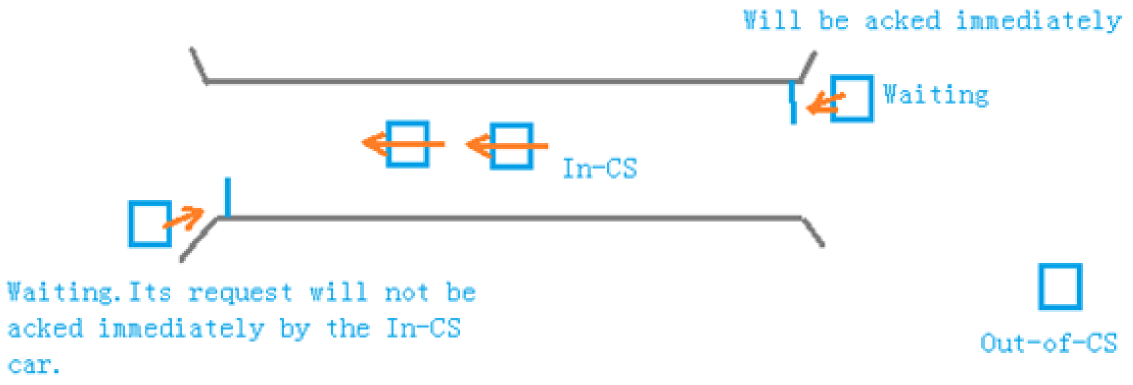

Will be acked immediately
Waiting
In-CS
Waiting. Its request will not be acked immediately by the In-CS car.
Out-of-CS

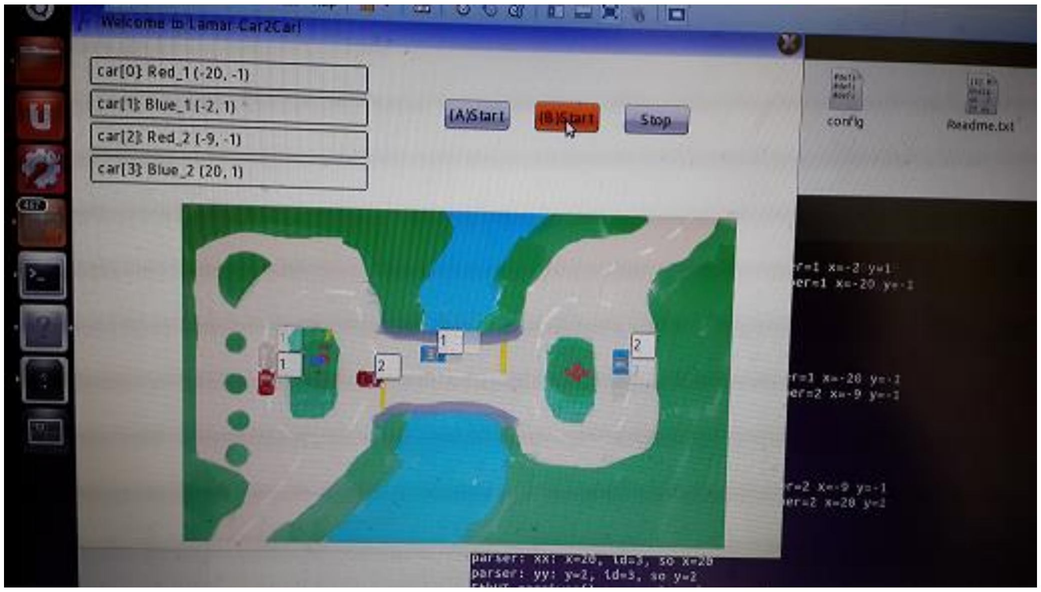
Fig-B Multiple cars can be on the bridge as long as they move in the same direction

How to compile and run this program

[1] How to compile and execute? Download this project to your Ubuntu machine, 'cd' into this directory and sh ./build.sh If everything is normal, you should see a GUI window appears.

[2] What to do next? Press button 'Start(A)' and see the cars run in circles and try to cross the bridge occasionally and press button 'Stop'. Then try another button 'Start(B)' to see part (b) for another situation.

Snapshot



```
car[0]: Red_1 (-20, -1)
car[1]: Blue_1 (-2, 1)
car[2]: Red_2 (-9, -1)
car[3]: Blue_2 (20, 1)
```

```
r=1 x=-2 y=1
er=1 x=-20 y=-1

r=1 x=-20 y=-1
er=2 x=-9 y=-1

r=2 x=-9 y=-1
er=2 x=20 y=2
parser: xx: x=20, id=3, so x=20
parser: yy: y=2, id=3, so y=2
```

More explanations about the source code

The source code for the ‘Car’ is under the sub-directory ‘car’ and source code for user interface is under the sub-directory ‘gui’. These two parts of programs are totally independent. The GUI application receives heartbeat report messages from cars and displays the information on screen.

Some extra code is for debugging and testing. They are also submitted because they are meaningful to show the independency of the car program and its GUI application. Another application called ‘simulator’ is actually using 4 threads and one mutex to perfectly represent the same scenario of part (a), one car is allowed to cross the bridge, which can be used to compare with the distributed version. Another ‘test’ application is to run the cars without GUI but only command line prints, which was used for debugging and now is a key proof for the independency of car program.

FTK is a 3rdparty GUI system. For it is cross-platform and on Linux it is based on X-window and you can see my '-lX11' in the Makefile under gui subdirectory. Even I was familiar with this GUI system, it still cost me much time to port it on my work station and develop my own car GUI application based on it. Oh, if you see some displaying flaws on GUI widgets, they are the problems embedded in the GUI itself.

Why displaying 'Welcome to Lamar Car2Car' on the title of GUI application? It is for fun and a sincere greeting to the real Car2Car research program from Europe: www.car-2-car.org

This program can only be executed on Ubuntu rather than other Linux machines, unfortunately. Mostly it is because the dependencies of the GUI system are slightly but terribly different on Linux distributions. It now has to run on Ubuntu is because my work station happens to be Ubuntu 12.04 64-bit LTS. The dependencies have to be checked and the Makefile should be slightly modified if you try to run on CentOS and other machines.