Monitoring the Number of Vacancies in a Car Parking by Using Distributed Mutual Exclusion

Thida Kyaw
Computer University (Taung-Ngu)
billytalent543@gmail.com, cutgo2009@gmail.com

Abstract

A number of solutions have been proposed for the problem of mutual exclusion in distributed systems. This paper is intended to simulate a system monitoring the number of vacancies in a car parking using Ricart-Agrawala's Distributed Mutual Exclusion Algorithm. In this proposed system, two entrances and two exits will be used. There will be a process at each entrance and exit that tracks the number of vehicles entering and leaving. Each process keeps a count of the total number of vehicles within the car parking and displays whether or not it is full. This system will be implemented to be safety (without allowing cars to enter when there is no vacancy), liveness (without taking long-time to decide whether or not more cars are allowed to enter the car parking) and concurrency (requests will be made by different processes at any time).

1. Introduction

A distributed system is one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages, and has no global control. In a distributed system, processes are scattered over network on each parallel separate computers but sharing some particular resources [3].

One of the fundamental problems in the distributed system is the shared resource management. Some resources can be shared by many users at a time (i.e. weather reports), while others (e.g. runway, taxi ways, etc) need to be shared in sequential order, by one user at a time. Thus, a safety critical system-of-systems must guarantee mutual exclusion. Coordination among users must be enforced through a communication and synchronization mechanisms to ensure the proper and correct use of the resource. Both centralized and distributed approaches are possible but they each have advantages or disadvantages.

A distributed system has no common memory and no common clock [4]. Therefore, it is sometimes impossible to say which of two events occurs first. To recap, a distributed system can be looked upon as a collection of computers separated in space, and not sharing a common memory that are not applicable to distributed systems.

In this paper, the problem of the absence of shared memory is solved by using multicast message-passing techniques. The problem of the absence of global clock is solved by using the Lamport's logical clock.

2. Related Work

The related works of distributed mutual exclusion are discussed in this session.

R.Atreya and N.Mittal provide a distributed algorithm for solving the group mutual exclusion problem based on the notion of surrogate-quorum. Intuitively, their algorithm uses the quorum that has been successfully locked by a request as a surrogate to service other compatible requests for the same type of critical section. Simulation results indicate that their algorithm outperforms the existing quorum-based algorithms for group mutual exclusion by as much as 50% in some cases. To measure the performance of a group mutual exclusion algorithm, they used the metrices such as message complexity, synchronization delay, waiting time, message overhead and concurrency [1].

P.K.Mohan and N.Mittal proposed an efficient algorithm for solving the problem when a relatively small number of groups are requested more frequently than others. Our algorithm has a message complexity of $2n - 1$ per request for critical section, where n is the number of processes in the system. Their objective is to develop an efficient distributed algorithm for solving the group mutual exclusion problem that exploits the fact that group access requests may not be uniformly distributed. Their algorithm is especially suited for applications in which a relatively small number of groups are requested more often than other groups. Their experiments indicated that their algorithm has much better performance than existing algorithms when group access is non-uniform [6].

The problem of defining distributed mutual exclusion algorithms to support multiple requests per-node has been solved by extending currently existing distributed mutual exclusion algorithms. E.Guadalupe, F.Meza, J.Perez and J.Piquer presented a generic framework to solve this problem as an

extension of a distributed mutual exclusion algorithm and a local mutual exclusion module. These two components are integrated through an extension key and the definition of clear communication interfaces for the key. The interaction between the distributed mutual exclusion algorithm and the mutual exclusion module is transparent, therefore these two components are not co-depending. The main idea is to maintain transparency in the interaction between the components, so the involved algorithms are not aware of the existence of the other one. The purpose is to preserve correctness properties of the original distributed mutual exclusion algorithm, implying that the extended algorithm is also correct [5].

3. Background Theory

3.1 Event Ordering

Since we are considering only sequential processes, all events executed in a single process (car) are totally ordered to use car parking. Also, a process must wait only after other process. Therefore, we define the event ordering:

1. If A and B are events in the same process, and A was executed before B, then A->B.
2. If A is the event of landing signal by one process and B is the event of see this landing signal, then A->B.
3. If A->B and B->C then A->C.

To determine than an event A happened before an event B, we need either a common clock or a set of perfectly synchronized clock. We associate with each system event a timestamp. For every pair of events A and B, if A->B, then the timestamp of A is less than the timestamp of B [2].

3.2 Synchronization

Synchronization is communication between processes taken place by calls to send and receive primitives. Concurrent processes should have synchronization between them. The degree of synchronization accuracy that is practically obtainable fulfils many requirements but is nonetheless not sufficient to determine the ordering of an arbitrary pair of events occurring at different computers. Lamport clocks are counters that are updated in accordance with the happened-before relationship between events. Vector clocks are improvement on Lamport clocks, because it is possible to determine by examining their vector timestamps whether two events are ordered by happened-before or are concurrent [4].

3.3. Mutual Exclusion

Shared resources need to be accessed exclusively. Part of process where shared resources are accessed is called critical section. A critical section (CS) is a piece of code where we want mutual exclusion. Mutual exclusion ensures that more than one concurrent process make a serialized access to critical section. There are many requirements on mutual exclusion, they are:

- At most one process in CS
- No process outside CS should prevent another processes from entering the CS
- No process should be blocked forever
- The solution should work independent of the relative speeds of the processes [7].

3.4. Distributed Mutual Exclusion (DME)

DME coordinates software on different computers, they agree upon assigning a resource or section of code to one particular task. In a distributed system, neither shared variables nor a local kernel can be used. Mutual exclusion has to be based exclusively on message passing [2].

3.4.1 Requirements for Distributed Mutual Exclusion

- Safety: at most one process can be in the critical section
- Liveness: a process requesting entry to critical section will eventually succeed
- Fairness: entry to critical section must respect to the order of request [2]

3.5 TimeStamp

Lamport's "logical clock" is used to generate timestamps. Lamport invented a simple mechanism, called a logical clock. A Lamport logical clock is a monotonically increasing software counter, whose value need bear no particular relationship to any physical clock. Each process pi keeps its own logical clock, Li, which it uses to apply so-called Lamport timestamps to events. The timestamp of event e at pi is denoted by Li(e). The timestamp of event e at whatever process is denoted by L(e) [8].

To capture the happened-before relation →, processes update their logical clocks and transmit the values of their logical clocks in messages as follows:

- LC1: Li is incremented before each event is issued at process pi :

  $L_i := L_i + 1$

- LC2:

  (a) When a process p_i send a message *m*, it piggybacks on *m* the value $t = L_i$.

  (b) On receiving (m,t), a process pj Computes $L_j := \max(L_j,t)$ and then applies LC1 before timestamping the event receive(*m*) [2].

3.5.1 Example of Using Logical Clocks

In Figure1, each processes P1, P2 and P3 has its logical clock initialized to 0. The clock values given are those immediately after the event to which they are adjacent.

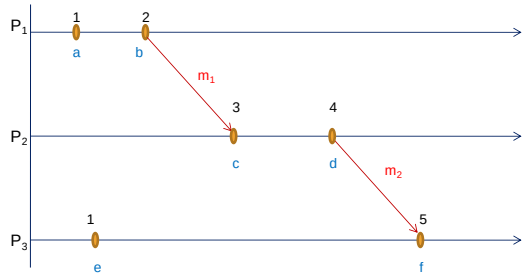

Figure 1. Example of using Logical Clock

4. Proposed System

The overview of the proposed system is shown in Figure 2. In this proposed system, two entrances and two exits will be used. There will be a process at each entrance and exit that tracks the number of vehicles entering and leaving.

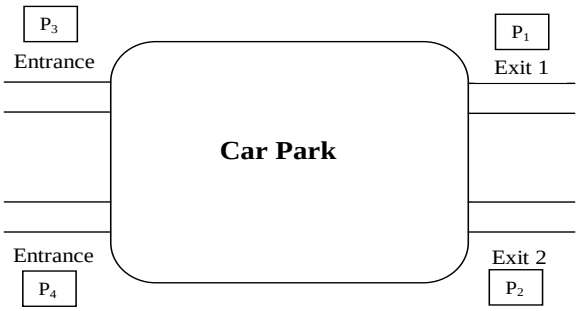

Figure 2. Overview of the Proposed System

Processes **P1, P2, P3 and P4** are responsible for calculating the number of vacancies in the car parking. Updating the number of vacancies is the **critical section** of these processes. The procedure for each process is as follows:

Processes P3, P4

- When cars want to enter the car parking, they check vacancies, request the rest processes and wait until all replies are received
- decrease the number of vacancies by 1 and let the car to enter
- multicast the resulted number of vacancies to all processes
- reply all requests stored in local queue
- exit the critical section

Processes P1, P2

- When cars leave the car parking, they request message to the rest processes
- wait until all replies are received
- increase the number of vacancies by 1 and let the car exit
- multicast the resulted number of vacancies to all processes
- reply all requests stored in local queue
- exit the critical section

4.1 Algorithm of the Proposed System

The algorithm for the proposed system is described as follow. This algorithm is based on Ricart and Agrawala's algorithm.

On initialization
state := RELEASED;
To enter the section
state := WANTED;
check type of process;
if (entering process) check vacancies
if (vacancies>0)
Multicast request to all processes;
T := request's timestamp;
Wait until (number of replies received = (N – 1));
state := HELD;
decrease the number of vacancies by 1;
multicast updated message to all processes;
else
display unavailable message;
else
if(leaving process)
Multicast request to all processes;
T:=request's timestamp;
Wait until (number of replies received = (N – 1));
state := HELD;
increase the number of vacancies by 1;
Multicast updated message to all Processes;
On receipt of a request <Ti, pi> at pj (i ≠ j)
if (state = HELD or (state = WANTED and (T, pj) < (Ti, pi)))
then
queue request from pi without replying;
else
reply immediately to pi;
To exit the critical section
send update information to all process in the system
state := RELEASED;
reply all requests stored in local queue;

Figure 3. Algorithm of the proposed system

5. System Implementation

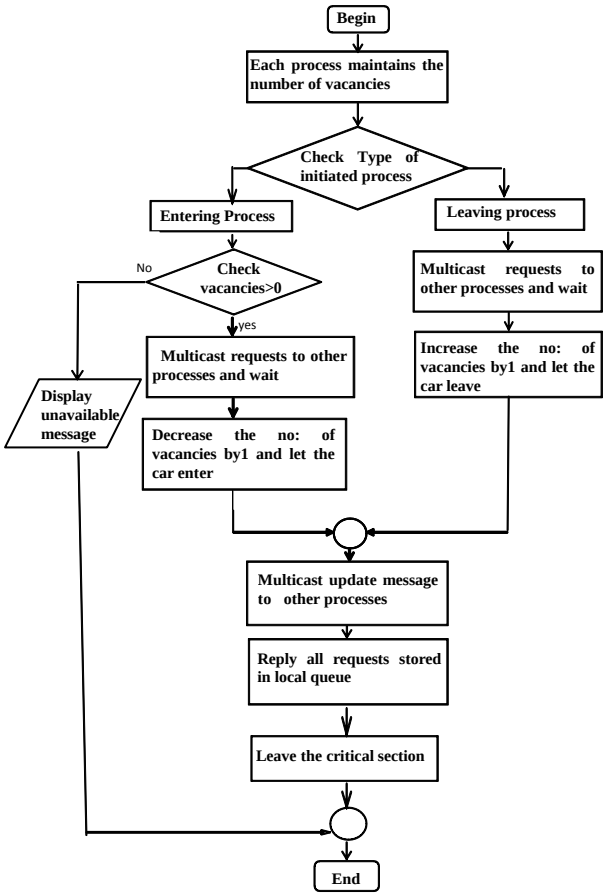

Figure 4. System Flow Diagram

In this system, processes at each exit and entrance maintain the number of vacancies in the car parking. Firstly, the process checks the type of process (entering or leaving) requested for updating the number of vacancies (critical region).

If the type of process is "Entering process", the process will check any vacancies in the car parking. If there is no vacancy, the system will show the "unavailable message" to the user. Otherwise, the process will multicast requests to all other processes and wait for their replies. After getting all reply, the process decreases the number of vacancies by1 and lets the car enter. Then it multicasts the updated information to all other processes. And it replies all requests stored in the local queue. Finally, the process leaves the critical section.

If the type of process is "Leaving process", the process makes requests to all other processes and waits for their replies. After getting all replies, the process increases the number of vacancies by1 and lets the car leave. Then it multicasts the updated information to all other processes. And it replies all requests stored in the local queue. Finally, the process leaves the critical section.

6. Conclusion

In this system, we present a system monitoring the number of vacancies in a car parking using Ricart-Agrawala's Distributed Mutual Exclusion algorithm. Our system will allow to process the car with lesser arrival time. Each entry can access only one car at a time. If two cars arrive at both of the entries at the same time, this system will allow the car at entry1 because this system is set priority to the entry1.

And therefore, only one process will be allowed at a time to update the number of vacancies in the car parking so that all processes can access the correct number of vacancies. This system solves the critical section problem by communicating processes among themselves, eliminating the need for a separate server.

7. References

[1]. R.Atreya and N.Mittal, "A Dynamic Group Mutual Exclusion Algorithm using Surrogate-Quorums".

[2]. G.Coulouris, J.Dollimore, T.Kindbrge, "Distributed Systems Concepts and Design" Third Edition, ISBN 0201-61918-0, 2001.

[3]. P.Eles , "Distributed System (TDDB97)", Lecture Notes, Institute of Datavetenskap IDA, Linkopings University, http:// www. ida.liu.se ~TDDB97.

[4]. J.Ganguly and M.D.Lemmon, "Theory of Clock Synchronization and Mutual Exclusion in Networked Control Systems", Technical Report of the ISIS Group at the University of Notre Dame ISIS-99-007), August, 1999.

[5]. E.Guadalupe, F.Meza, J.Perez and J.Piquer , "A Framework to Extend Distributed Mutual Exclusion Algorithms to Support Multiple Requests Per-Node".

[6]. P.K.Mohan and N.Mittal, "An Efficient Distributed Group Mutual Exclusion Algorithm for Non-Uniform Group Access".

[7]. S.Silberschatc , V.Galvin, A.Gagne,"Operating System Concepts" Sixth Edition, ISBN 0-471-41743-2, 2002.

[8]. S.A.Tanenbaum, "Distributed Operating Systems", Person Education, Inc. ISBN 81-7808-294-2, 1995.