Evaluation of Distributed Mutual Exclusion through Practical Simulation

Maekawa’s Algorithm

Jordan Adamek
Computer Science Department
Kent State University
Kent, OH 44242

Abstract—I implement a simulation engine which measures the practical performance of Maekawa’s permission-based distributed mutual exclusion algorithm. I measure the latency, efficiency, and correctness of the algorithm with varying system load, as well as the number of critical section requests granted out of time stamp order. I lastly report on my own coding techniques employed to construct the engine.

I. INTRODUCTION

The distributed mutual exclusion problem is the mutually exclusive access of some resource between the processes that share it. A solution to this problem must meet two specifications: no two processes that share a resource may access that resource simultaneously (safety); also, a process which requests access to a shared resource must be allowed to access that resource eventually (liveness). One such solution to this problem through permission-based protocol in decentralized systems was proposed by Mamoru Maekawa, and uses a number of messages proportional to root N (where N is the number processes) [1]. The root N derives from the use of quorums; a set of process ids (integers). A process only needs the permission of the members of its quorum, to access a shared resource (critical section). Critical section request and access are resolved by three major message types: Request, Grant, and Release. An additional three are required to avoid potential deadlock that may otherwise occur: Inquire, Failed, and Yield [2]. While there this algorithm’s correctness is theoretically proven, little research has been done to test its practical performance is a realistic programming environment.

In this research study, I implement the algorithm in a simulation engine in guarded command notation and test its practical performance. I simulate critical section request (system load), and vary the number of requests sent per computation.

A. Quorum Construction

The construction of quorums in non-trivial; a process’s quorum must satisfy two properties: inclusion and intersection. Inclusion specifies that the id of the process which holds the quorum must be a part of its own quorum. Intersection states that for any two processes in the system, their respective quorums must intersect by at least one value. Two other desirable properties are uniqueness – no two quorums are exactly the same, and anti-uniqueness – no two quorums have equivalent anti-quorums. One method for quorum construction was proposed my Maekawa which places all process ids into a square grid to guarantee these four properties among all system quorums. This method constructs quorums of size 2 root N (the combined length of a column and row in the grid). Another method, called billiard quorums, reduces quorum size to root 2 root N [3].

B. Guarded Command Notation

The algorithm was implanted in guarded command notation. Executed code is specified as a set of guarded actions belonging to each process in the system. An action has two parts: a guard and a command. The guard of an action is a Boolean predicate on the local variables of the process and its neighbors (in this case quorum members) and the process’s own private variables. An action is enabled if its guard evaluates to true, and is disabled otherwise. The command of an action is a set of executable statements that update or assign new values to the process’s private and local variables. In order to implement the algorithm in a real programming environment, I interpreted it into guarded command notation. Critical section request was implemented through setting a Boolean local variable Request, to true; critical section access was implemented by execution of a public method executeCS.

II. SIMULATION ENGINE

The engine simulated asynchronous computations of Maekawa’s algorithm by iterating a set number of computation states (global states). Each state, the engine queried all system processes for enabled actions. Of the enabled processes, a random one was selected to execute the guarded action’s command. If more than one action was enabled for the chosen process, a random action of the enabled was selected for execution. This effectively simulated interleaving execution semantics (a single process executes per program state).

The engine simulated network load (number of CS requests) through a request scheduling protocol. An integer array of size S (number of maximum states in a single computation) has its elements randomly incremented a set number of times equal to the desired system load. At the beginning of each state throughout the computation, a number of currently non-requesting processes are randomly chosen and
set Request, to true. The number chosen is equal to the value at the i’th element in the scheduling array (i being the current state number). Note, in this model, a single process may request critical section more than once throughout a single computation. A computation iterates the state number until either it reaches S or deadlock. In the case of deadlock, such an event is immediately reported and a snapshot of the program state saved for later diagnostic viewing.

A. Measurements and Metrics

The simulation engine measured and tracked several performance variables of system processes, recording the resulting metrics in array-like objects called DataSets (floating-point arrays with methods to calculate statistical data on the fly). These metrics included: latency – the number of program states from a given CS request to its execution, throughput (efficiency) – the number of CS accesses per guarded action executed, the number of safety violations – for some process whose CS-access action is enabled, a member of its quorum also has such action enabled, and the number of CS accesses granted out of timestamp order.

The procedure to measure each is as follows. For latency, a secondary integer array of size N (number of system processes) is assigned initial values of zero. When a process requests critical section, it sets the j’th element of the array (where j is the process’s id) to the current state number. When a process accesses the CS, a latency measurement is recorded as the difference in the current state number and the value of the accessing process’s j’th element in the integer array. Through-out is simply recorded as the total number of CS accesses divided by executed guarded commands (for Interleaving, this is equal to the number of computation states). Safety violations are also a straightforward measurement; I was more concerned of the existence of any such violation, rather than the exact quantity. The number of CS accesses granted out of timestamp order was recorded as follows: An integer lastTS tracks the timestamp of the last granted CS access. When a processes enters the CS, if its current timestamp is less than lastTS, increment the number of measured out of order grants. Finally, at the end of the computation, record the total number.

III. EXPERIMENTAL PROCEDURE

In this study, I simulated a system side of 40 processes, with a fixed quorum size of 5. These values stem from my using the billiard quorum method of construction; quorums were fixed throughout all computations. I varied the system load from one CS request, to the number of system processes. For each load value, I simulated 10,000 computations with a maximum state limit of 1,000 program states.

IV. RESULTS

Of the original four metrics measured, safety violations and CS accesses grant out of order results in zero across the board. The former implies that Maekawa’s original algorithm and my implementation of it are correct. The latter implies that no CS grant was ever made out of timestamp order. The measurements for latency and throughput are shown in Figures 1 and 2, respectively.

V. RESULTS ANALYSIS

The measurements for the latency as a function of system load were somewhat expected; as the system becomes more crowded with CS requests, the number of states required to enter CS increases. Also, the relationship of the two is expectedly non-linear. As the system load increases, the probability that two requesting processes belong to the same quorum increases. Not only does the probability increase, but the number of sites at which this may occur increases as well. A similar argument explains the relation found in throughput, but in this case we see a decrease: the number of actions required to execute critical section increases similarly to latency, while the number of CS accesses is approximately linear and equal to the system load. Keep in mind a CS request may not be satisfied within the 1,000 state computation limit. The non-existent results for out of order CS access grants, however, were not expected. Neither Maekawa’s original paper nor the applied fix for deadlock avoidance in any way guaranteed that CS access would be granted in time stamp order. In fact, were such a property exhibited, there would be
no need for the three deadlock-avoidance message types. As one of the last remaining fixed variables, it is quite possible that quorum construction is at the root of this unexpected result. The method of billiard quorum construction may somehow guarantee that ordered CS access. Whether or not Failed and Inquire messages are being sent however, is left to later research.

VI. CONCLUSION

The simulation of Maekawa’s algorithm in a practical setting proved worthwhile, in that a theoretically unexpected result was uncovered. While latency, throughput, and safety measurements proved to match the theoretical and hypothetical behaviors, the number of out of order CS grants was entirely unexpected; namely, there were none. This suggests that billiard quorum construction for a system size of 40 processes and fixed quorum size of 5 retains ordered CS execution. It may possibly suggest that such a system configuration could abandon the need for the deadlock avoidance message types; however, this would require further investigation.

VII. FUTURE WORK

Given the unexpected result discovered in this study, further investigation into the practical implementation of Maekawa’s algorithm should be taken. Chiefly, different methods of quorum construction should be tested to confirm or deny the influence of this parameter over the number of CS accesses granted out of time stamp order, among other parameters. Other investigations to consider would include scalability tests, as well as performance under different execution semantics such Synchronous or Power Set.

VIII. CODE DESCRIPTION

The simulation was coded in C++ programming language using Visual Studio 2010 (VC++9). The engine was comprised of four major base classes: Action, Process, Network, and Engine. Each class was extended for the purpose of implemented Maekawa’s algorithm specifically. Four additional supplementary classes were implemented as well: Message, Channel, Quorum, and DataSet. Each guarded action was implemented as an extension of the base Action class, overriding two virtual methods, Boolean method Guard and void method Command, to match the intended guarded action they represented. Quorums contained a set of const Process pointers rather than ids for quick and easy access to process variables. I used class co-dependence to create an intuitive relationship between Action and Process classes: the Process class contains a set of const Action pointers to the actions it defines, and the Action class contains a variable process, a non-const Process pointer to the process that defines it. Implementing things way, processes could easily iterate over defined actions without having necessarily know anything about their implementation; the Guard and Command overrides handled the difference in action execution logic. Also, on the Action side, these overridden commands were able to reference their defining process without iteration or search over the network, or any parameter pass. They already know which process they refer to.

The extended MaekawaNetwork class implemented Billiard quorum construction and process initialization. Each process was first added to the network into a node vector. Then, a setupQuorums method actually constructed a quorum based on the broken billiard paths described in the literature referencing the node vector directly by process id, calling a public attach method which adds the target process (passed as an argument) to the process’s quorum.

The extended MaekawaEngine class implemented the metric measurement logic: request scheduler, timestamp recorder, request recorder, and the four DataSets objects assigned for each metric. DataSet was implemented simply as a map of floating point integers (x-y pairs), and contained methods for reporting statistical average and margin of error.

ACKNOWLEDGMENT

Utmost thanks to Mikhail Nesterenko, whose guidance and advice in past experience with distributed algorithm simulation provided an elementary starting point for this project. I would also think him for the continued advice in the suggestion of billiard quorums as a construction method, and his overall assistance in the understanding of the algorithm.

REFERENCES

