

Network Resource Allocation Mutual Exclusion

Bernhard Aichinger

Peter Praxmarer

February 27, 2002

Introduction

- The Mutual Exclusion problem in general subsums any problems regarding the concurrent use of a single resource by many processes
- We present two algorithms that solve the problem with Lamport's LogicalTime concept:
 - Logical Time Mutual Exclusion
 - Ricart Agrawala Mutual Exclusion

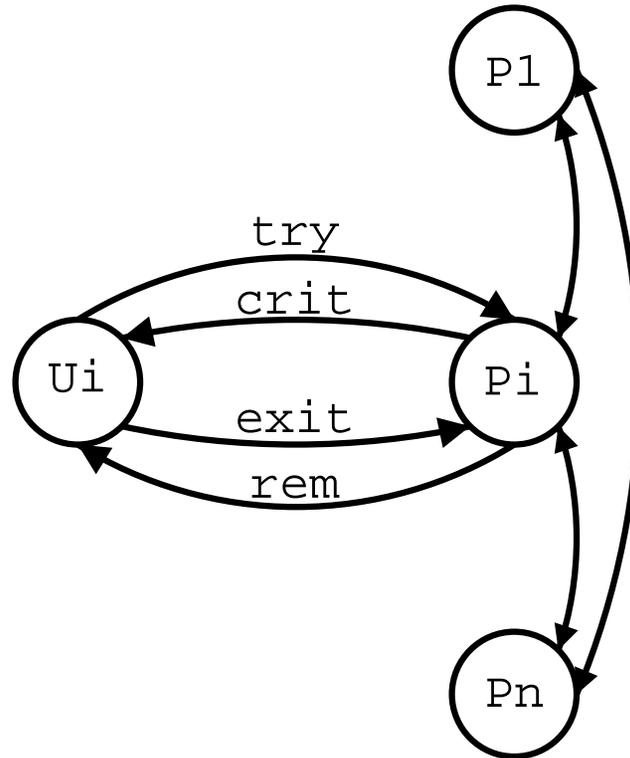
Assumptions

- Asynchronous Network System
- Communication via reliable FIFO channels
- Algorithms can be used in:
 - Singlecast/Broadcast systems (broadcast can be emulated by point to point channels)
 - Broadcast only systems

The Problem (1)

- n users, U_1, \dots, U_n , defined to be I/O automata
- The *system* A being used to solve the problem in an asynchronous network system
- Process P_i corresponds to user U_i
- $try_i, crit_i, exit_i$ and rem_i are used for communication between U_i and P_i
- Communication system contains a combination of send/receive and broadcast channels

The Problem (2)



Interactions between components for the mutual exclusion problem

Correctness Conditions (1)

Mutual exclusion: There is no reachable system state in which more than one user is in the critical region C

Progress: At any point in a fair execution,

1. If at least one user is in T and no user is in C , then at some later point some user enters C
2. If at least one user is in E , then at some later point some user enters R

Correctness Conditions (2)

Lockout-freedom: In any fair execution, the following hold:

1. If all users always return the resource, then any user that reaches T eventually enters C
2. Any user that reaches E eventually enters R

Well-formedness: In any execution and for any i , the subsequence describing the interaction between U_i and A is well-formed for U_i

Mutual exclusion - *LogicalTimeME* (1)

- Generates logical times for events using the *LamportTime* strategy
- Logical time is a pair (c, i) , where $c \in N$ and i is a process index
- Logical time pairs are ordered lexicographically
- Broadcast and send/receive communication between processes

Mutual exclusion - *LogicalTimeME* (2)

- Each process P_i maintains a single history data structure
- For each j , $history(j)_i$ records all the messages P_i has ever received from P_j
- The *try* and *exit* messages are broadcasted
- A *try* message is acknowledged by an *ack* message.
- P_i can perform a $crit_i$ when its latest *try* request has reached its $history(i)$

Mutual exclusion - *LogicalTimeME* (3)

- Every other request that P_i has heard of with a smaller logical time has already been granted
- P_i has received a message with a greater logical time from every other process
- P_i can perform a rem_i as soon as its latest *exit* request has reached its $history(i)$

The Proof - Mutual Exclusion (1)

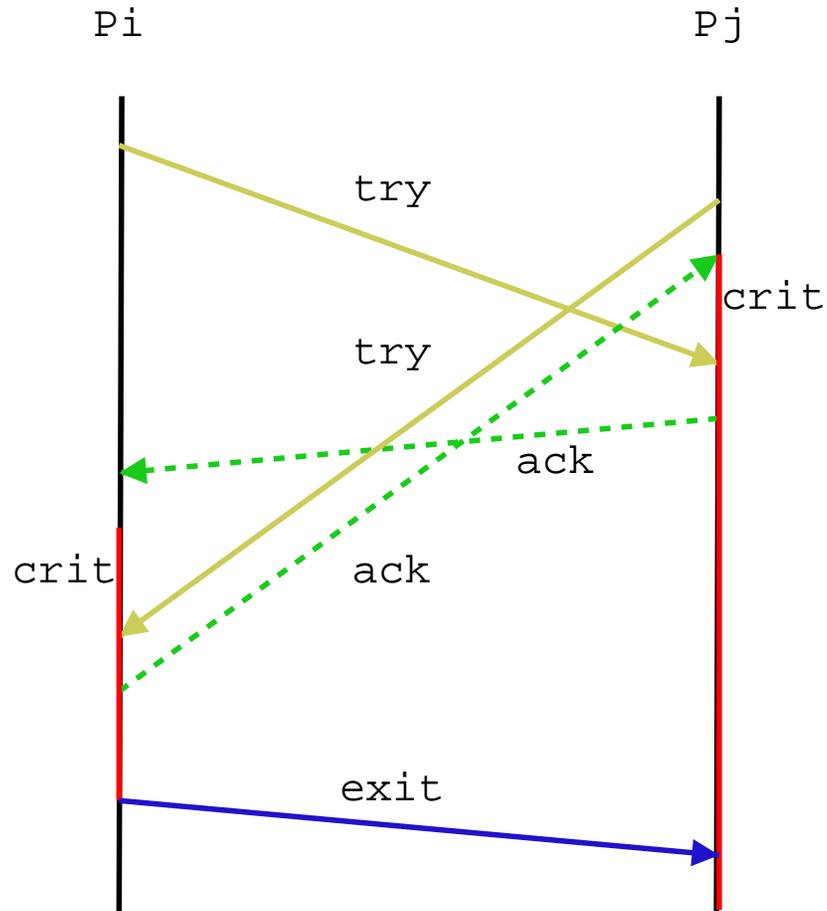
To see that the algorithm guarantees mutual exclusion, we proceed by contradiction:

- Suppose two processes P_i and P_j are in C at the same time
- $t_i < t_j$ (latest *try* message of P_i and P_j)
- In order to perform *crit* _{j} and enter C , P_j had to see a message from P_i with logical time *greater* than t_j and hence *greater* than t_i

The Proof - Mutual Exclusion (2)

- FIFO property implies that P_j must have seen P_i 's *try* message when it performed *crit_j*.
- *crit_j* implies that P_j must have seen a subsequent *exit* message from P_i .
- This implies that P_i must have already left C at the time P_j performed *crit_j*.
- → Contradiction!

The Proof - Mutual Exclusion (3)



Impossible communication scenario.

The Proof - Lockout-Freedom (1)

- Lockout-Freedom implies system progress.
- All the preconditions for $crit_i$ must eventually become satisfied, because
 - if P_i is in region \mathbf{T} and has a *try* message with the *smallest* logical time t_i among those for current requests
 - then *fair execution* implies that eventually P_i receives its own *try* message and places it in $history(i)_i$.

The Proof - Lockout-Freedom (2)

- Also since *try* message receive corresponding *ack* messages and the *clock* variables are managed using the LT discipline, P_i eventually receives a message from each of the other processes with LT *greater* than t_i .
- Finally, since P_i 's request is the current request with the *smallest* LT, *any* request with a smaller LT *must* have already had a corresponding *exit* event. (Delivery of the messages is implied by the fairness properties of the broadcast channel.)

Complexity Analysis (1)

Communication complexity For every request:

- 1 *try* broadcast - (n individual messages)
- $n - 1$ *ack* messages in response to the *try* message
- 1 *exit* broadcast - (n individual messages)
- Total amount of messages: $3n - 1$ messages

Complexity Analysis (2)

Time complexity Time from try_i to $crit_i$

- Strongly isolated request (best case)
- No residual messages arising when try_i event occurs
- Time between try_i to $crit_i$: $2d + O(\ell)$
- d is the upper bound on the delivery of any message and ℓ is the upper bound on time for each process task

Improvements to *LogicalTimeME* - *RicartAgrawalaME*

- Simple variation on the *LogicalTimeME* algorithm
- Reduces the communication complexity
- Needs only $2n - 1$ messages per request
- Improves *LogicalTimeME* by acknowledging requests in a careful manner
- Eliminates the need for *exit* messages
- Uses broadcast and send/receive communication

Mutual Exclusion - *RicartAgrawalaME*

- Logical time events are generated as in *LogicalTimeME*
- Only two messages: *try* and *ok*, each carries the *clock* value
- After a *try_i*, P_i broadcasts *try* messages just as in *LogicalTimeME* and can go to C after it receives subsequent *ok* messages
- Interesting part of the algorithm is a rule for when a process P_i can send an *ok* message to another process P_j

RicartAgrawalaME - *ok* Messages (1)

The idea is to use a priority scheme. In response to a *try* message from P_j , P_i does the following:

- If P_i is in E or R , or in T prior to broadcasting the *try* message for its current request, then P_i replies with *ok*
- If P_i is in C , then P_i defers replying until it reaches E and then immediately sends any deferred *oks*

RicartAgrawalaME - *ok* Messages (2)

- If P_i is in T and its current request has already been broadcasted, then P_i compares the logical time t_j associated with the incoming *try* message of P_j :
 - If $t_i > t_j$, then P_i 's own request is given lower priority and P_i replies with an *ok* message
 - Otherwise, P_i 's own request has higher priority, so it defers replying until it finishes its next critical region. At that time, it immediately sends any deferred *oks*
 - P_i can perform a rem_i at any time after it receives an $exit_i$

The Proof - Mutual Exclusion (1)

To see that the algorithm guarantees mutual exclusion, we proceed by contradiction:

- Suppose two processes P_i and P_j are in C at the same time
- $t_i < t_j$ (latest *try* message of P_i and P_j)
- There must have been *try* and *ok* messages sent from each P_i and P_j to the other
- At each process the receipt of the *try* message precedes its sending of the corresponding *ok*
- Several possible orderings of the various events

The Proof - Mutual Exclusion (2)

- Claim that $t_j < r_i$ and $r_i < t_i$
- $\Rightarrow t_j < t_i$ is a contradiction to the assumption
- Therefore, at the time P_i receives P_j 's *try* message, P_i is either in T or in C
- P_i 's rules say that it should defer sending an *ok* message until it finishes its own critical region
- P_j could not enter C before P_i leaves \Rightarrow contradiction

Complexity Analysis (1)

Message complexity: For every event

- 1 *try* broadcast - (n individual messages)
- $n - 1$ *ok* messages
- Total amount of messages: $2n - 1$ messages

Complexity Analysis (2)

Time complexity: Time from try_i to $crit_i$

- Strongly isolated request (best case)
- No residual messages arising when try_i event occurs
- Time between try_i to $crit_i$: $2d + O(\ell)$
- d is the upper bound on the delivery of any message and ℓ is the upper bound on time for each process task