Distributed Systems

Rik Sarkar
James Cheney

Distributed Mutual Exclusion

February 10, 2014

Overview

- It is generally important that the processes within a distributed system have some sort of agreement
- Agreement may be as simple as the goal of the distributed system
  - Has the general task been aborted?
  - Should the main aim be changed?
- This is more complicated than it sounds, since all the processes must, not only agree, but be confident that their peers agree.
- In this part of the course we will examine how distributed processes can agree on particular values
- We will first look at **mutual exclusion** to coordinate access to shared resources

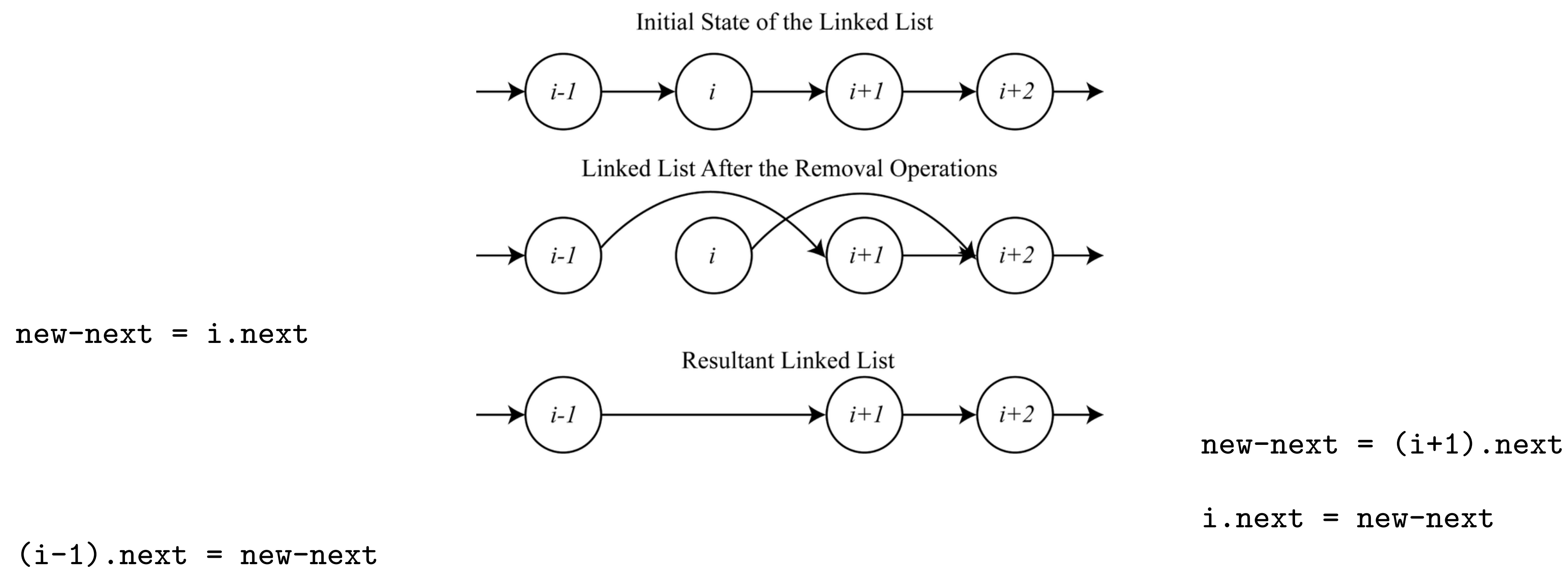
Mutual Exclusion

- Ensuring mutual exclusion to shared resources is a common task
- For example, processes A and B both wish to add a value to a shared variable ‘a’.
- To do so they must store the temporary result of the current value for the shared variable ‘a’ and the value to be added.

Time	Process A	Process B	
1	t = a + 10		A stores temporary
2		t’ = a + 20	B stores temporary
3		a = t’	(a now equals 25)
4	a = t		(a now equal 15)

- The intended increment for a is 30 but B’s increment is nullified

Concurrent updates



Shamelessly stolen from Wikipedia

- A higher-level example is concurrent editing of a file on a shared directory
- Another good reason for using a source code control system

Distributed Mutual Exclusion

- On a **single** system mutual exclusion is usually a service offered by the operating system's kernel.
  - Some languages also provide support for mutual exclusion
- In some cases the server that provides access to the shared resource can also be used to ensure mutual exclusion
- We will consider the case that this is for some reason inappropriate
  - the resource itself may be distributed for example
- For a **distributed** system we need a solution that operates only via message passing

Generic Algorithms for Mutual Exclusion

- We will look at the following algorithms which provide mutual exclusion to a shared resource:
    1. The central-server algorithm
    2. The ring-based algorithm
    3. Ricart and Agrawala — based on multicast and logical clocks
    4. Maekawa's voting algorithm
- We will compare these algorithms with respect to:
    - Their ability to satisfy three desired properties
    - Their performance characteristics
    - Their ability to tolerate failure

Assumptions and Scenario

- Assumptions:
  - The system is asynchronous
  - Processes do not fail
  - Message delivery is reliable: all messages are eventually delivered exactly once.
- Scenario: Assume that the application performs the following sequence:
  1. Request access to shared resource, blocking if necessary
  2. Use the shared resource exclusively — called the **critical section**
  3. Relinquish the shared resource

Assumptions and Scenario

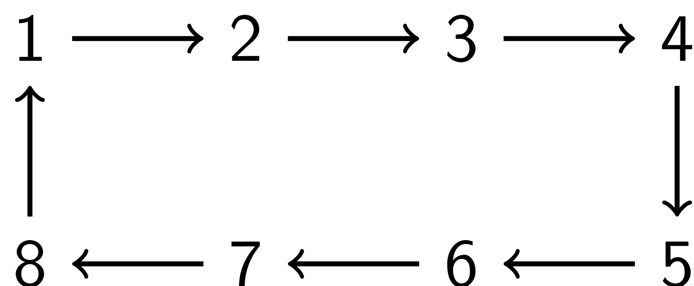
- Here we are considering mutual exclusion of a **single** critical section
- We assume that if there are **multiple** resources then either:
  - Access to a single critical section suffices for all the shared resources, OR
  - A process cannot request access to more than one critical section concurrently, OR
  - Deadlock arising from two (or more) processes holding each of a set of mutually desired resources is avoided using some other means
- We also assume that a process granted access to the critical section will eventually relinquish that access

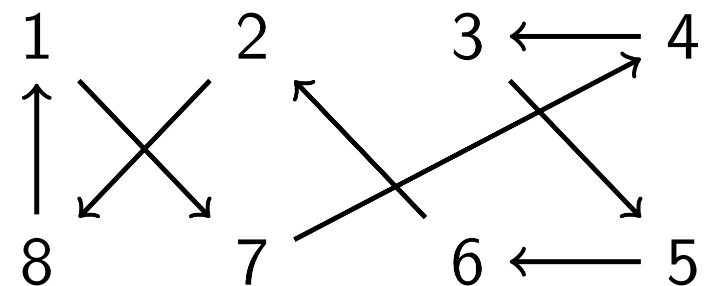
Central Server Algorithm

- The simplest way to ensure mutual exclusion is through the use of a centralized server
  - This is analogous to the operating system acting as an arbiter
- There is a conceptual **token**, processes must be in possession of the token in order to execute the critical section
- The centralized server maintains **ownership** of the token
- To **request** the token; a process sends a request to the server
- If the server currently has the token it immediately responds with a message, granting the token to the requesting process
  - When the process completes the critical section it sends token back to server
- If the server doesn't have the token, **some other process** is currently in the critical section
  - In this case the server queues the incoming request for the token
  - Responds to next request when the token is returned

Ring-based Algorithm

- Central server is a single point of failure
- A simple way to arrange for mutual exclusion without the need for a master process, is to arrange the processes in a logical **ring**.
- The ring may of course bear little resemblance to the physical network or even the direct links between processes.





Ring-based Algorithm

- The token passes around the ring continuously.
- When a process receives the token from its neighbor:
  - If it does not require access to the critical section it immediately forwards on the token to the next neighbor in the ring
  - If it requires access to the critical section, the process:
    1. retains the token
    2. performs the critical section and then:
    3. to relinquish access to the critical section
    4. forwards the token on to the next neighbor in the ring
- Token can get lost due to crash or message drop

Multicast and Logical Clocks

- Ricart and Agrawala developed an algorithm for mutual exclusion based upon **multicast** and **logical clocks**
- The idea is that a process which requires access to the critical section first broadcasts this request to all processes within the group
- It may then only actually enter the critical section once **all** of the other processes have granted their approval
- Of course the other processes do not just grant their approval indiscriminately
- Instead their approval is based upon whether or not they consider their own request to have been made first

Multicast and Logical Clocks

- Each process maintains its own Lamport clock
- Recall that Lamport clocks provide a partial ordering of events
  - that this can be made a total ordering by considering the process identifier of the process observing the event
- Requests to enter the critical section are multicast to the group of processes and have the form $\{T, p_i\}$
- T is the Lamport time stamp of the request and p_i is the process identifier
- This provides us with a total ordering of the sending of a request message $\{T_1, p_i\} < \{T_2, p_j\}$ if:
  $T_1 < T_2$ or $T_1 = T_2$ and $p_i < p_j$

Requesting Entry

- Each process retains a variable indicating its state, it can be:
  1. "**Released**" — Not in or requiring entry to the critical section
  2. "**Wanted**" — Requiring entry to the critical section
  3. "**Held**" — Acquired entry to the critical section and has not yet relinquished that access.
- When a process requires entry to the critical section
  - it updates its state to "Wanted" and multicasts a request to enter the critical section to all other processes. It stores the request message $\{T_i , p_i \}$
  - Only once it has received a "permission granted" message from all other processes does it change its state to "Held" and use the critical section

Responding to requests

- A process currently in the "Released" state:
  - can immediately respond with a permission granted message
- A process currently in the "Held" state:
  - Queues the request and continues to use the critical section
  - Once finished using the critical section responds to all such queued requests with a permission granted message
  - changes its state back to "Released"
- A process currently in the "Wanted" state:
  - Compares the incoming request message $\{T_j , p_j \}$ with its own stored request message $\{T_i , p_i \}$ which it broadcasted
  - If $\{T_i,p_i\} < \{T_j,p_j\}$ then the incoming request is queued as if the current process was already in the "Held" state
  - If $\{T_i,p_i\}>\{T_j,p_j\}$ then the incoming request is responded to with a permission granted message as if the current process was in the "Released" state

Maekawa's voting algorithm

- Maekawa’s voting algorithm improves upon the multicast/logical clock algorithm with the observation that not all the peers of a process need grant it access
- A process only requires permission from a **voting set** (subset) of all the peers, provided that the subsets associated with any pair of processes overlap
- The main idea is that processes **vote** for which of a group of processes contending for the critical section will be given access
- Processes within the intersection of two competing voting sets can only vote for one process at at time, ensuring mutual exclusion

Maekawa's voting algorithm

- Each process p_i is associated with a voting set V_i of processes
- The set V_i for the process p_i is chosen such that:

1. $p_i \in V_i$ — A process is in its own voting set
2. $V_i \cap V_j \neq \{\}$ — There is at least one process in the overlap between any two voting sets
3. $| V_i | = | V_j |$ — All voting sets are the same size
4. Each process p_i is contained within M voting sets

Maekawa's voting algorithm

- The main idea in contrast to the Ricart-Agrawala algorithm is that each process may only grant access to one process at a time
- A process which has already granted access to another process cannot do the same for a subsequent request. In this sense it has already voted
  - Those subsequent requests are queued
- Once a process has used the critical section it sends a **release** message to its voting set
- Once a process in the voting set has received a **release** message it may once again vote, and does so immediately for the head of the queue of requests if there is one

The state of a process

- As before each process maintains a state variable which can be one of the following:
  - “Released” — Does not have access to the critical section and does not require it
  - “Wanted” — Does not have access to the critical section but does require it
  - “Held” — Currently has access to the critical section
- In addition each process maintains a boolean variable indicating whether or not the process has “voted”
  - Voting is not a one-time action. This variable really indicates whether some process within the voting set has access to the critical section and has yet to release it
- To begin with, these variables are set to “Released” and False respectively

Requesting Permission

- To request permission to access the critical section a process p_i:
  - Updates its state variable to “Wanted”
  - Multicasts a request to all processes in the associated voting set V_i
- When the process has received a “permission granted” response from all processes in the voting set V_i :
  - update state to “Held” and use the critical section
- Once the process is finished using the critical section, it updates its state again to “Released” and multicasts a “release” message to all members of its voting set V_i

Granting Permission/ Voting

- When a process p_j receives a request message from a process p_i:
- If its state variable is "Held" or its voted variable is True:
  - Queue the request from p_i without replying
- otherwise:
  - send a "permission granted" message to p_i
  - set the voted variable to True

Granting Permission/ Voting

- When a process p_j receives a "release" message:
- If there are no queued requests:
  - set the voted variable to False
- otherwise:
  - Remove the head of the queue, p_q:
  - send a "permission granted" message to p_q
  - The voted variable remains True

Desirable properties

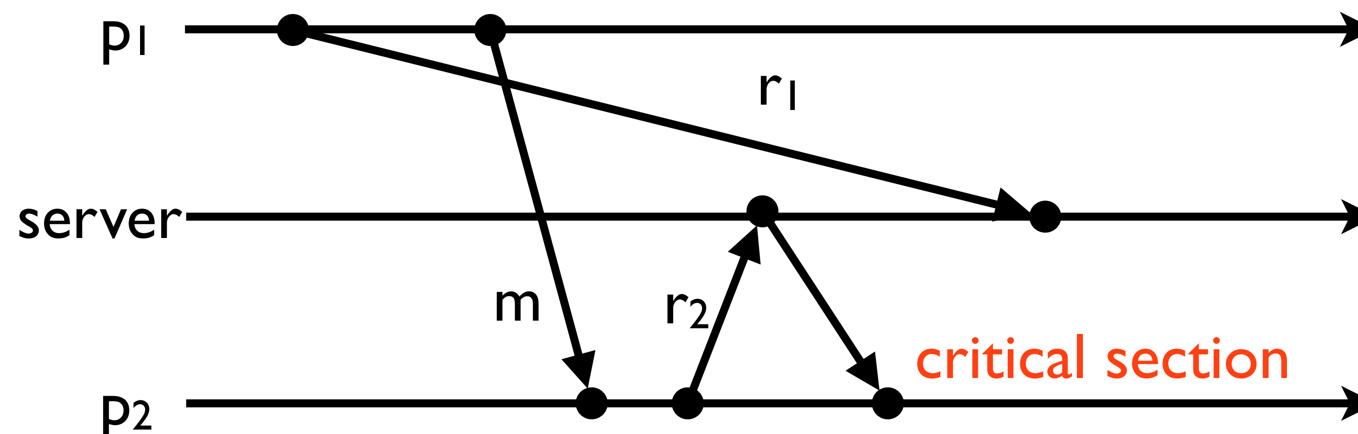
- **Safety:** At most one process may be in the critical section
- **Liveness:** Requests to enter and exit the critical section eventually succeed
  - The Liveness property assures that we are free from both deadlock and starvation
  - *starvation* is the indefinite postponement of the request to enter the critical section from a given process

Fairness

- **Fairness:** If e_1 and e_2 are requests to enter the critical section and $e_1 \rightarrow e_2$, then the requests should be granted in that order.
  - Note: our assumption of request-enter-exit means that process will not request a second access until after the first is granted
- Here we assume that when a process requests entry to the critical section, then until the access is granted it is blocked only from entering the critical section
  - In particular it may do other useful work and send/receive messages
  - If we were to assume that a process is blocked entirely then the Fairness property is trivially satisfied

Properties: Central Server Algorithm

- Given our assumptions that no failures occur it is straight forward to see that the central server algorithm satisfies the Safety and Liveness properties
- The Fairness property though is not satisfied
- Consider two processes p_1 and p_2 and the following sequence of events:

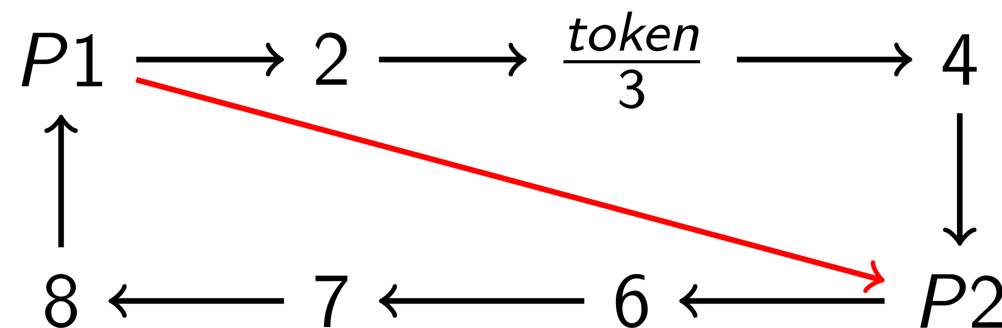


- Despite send(r_1) → send(r_2) the r_2 request was granted first.

Properties: Ring-based Algorithm

- It is straightforward to determine that this algorithm satisfies the Safety and Liveness properties.
  - assuming no failures...
- However once again we fail to satisfy the Fairness property

Ring algorithm — (Un)fairness



- Processes may send messages to one another independently of the token
- Suppose again we have two processes P_1 and P_2; consider the following events
    1. Process P_1 wishes to enter the critical section but must wait for the token to reach it.
    2. Process P_1 sends a message m to process P_2.
    3. The token is currently between process P_1 and P_2 within the ring, but the message m reaches process P_2 before the token.
    4. Process P_2 after receiving message m wishes to enter the critical section
    5. The token reaches process P_2 which uses it to enter the critical section before process P_1

Properties: Ricart and Agrawala

- Safety — If two or more processes request entry concurrently then whichever request bears the lowest (totally ordered) timestamp will be the first process to enter the critical section
    - All others will still be awaiting a permission granted message from (at least) that process until it has exited the critical section
- Liveness — Since the request message timestamps are a total ordering, and all requests are either responded to immediately or queued and eventually responded to, all requests to enter the critical section are eventually granted
- Fairness — Since Lamport clocks assure us that $e_1 \rightarrow e_2$ implies $L(e_1) < L(e_2)$:
    - for any two requests r_1, r_2 if $r_1 \rightarrow r_2$ then the timestamp for r_1 will be less than the timestamp for r_2
    - Hence the process that multicast r_1 will not respond to r_2 until after it has used the critical section
- Therefore this algorithm satisfies all three desired properties

Maekawa's algorithm — Deadlock

- The algorithm as described does not respect the Liveness property
- Consider three processes p_1, p_2 and p_3
- Their voting sets: $V_1 = \{p_1, p_2\}$, $V_2 = \{p_2, p_3\}$ and $V_3 = \{p_3, p_1\}$
- Suppose that all three processes concurrently request permission to access the critical section
- All three processes immediately vote for to their own requests
- All three processes have their “voted” variables set to True
- Hence, p_1 queues the subsequently received request from p_3
- Likewise, p_2 queues the subsequently received request from p_1
- Finally, p_3 queues the subsequently received request from p_2
- :(

Properties: Maekawa's algorithm

- Safety — Safety is achieved by ensuring that the intersection between any two voting sets is non-empty.
  - A process can only vote (or grant permission) once between each successive “release” message
  - But for any two processes to have concurrent access to the critical section, the non-empty intersection between their voting sets would have to have voted for both processes
- Liveness — As described the protocol does not respect the Liveness property
  - It can however be adapted to use Lamport clocks similar to the previous algorithm
- Fairness — Similarly the Lamport clocks extension to the algorithm allows it to satisfy the Fairness property

Performance Evaluation

- For performance we are interested in:
  - The number of **messages** sent in order to **enter** and **exit** the critical section
  - The **client delay** incurred at each entry and exit operation
  - The **synchronisation delay**, this is delay between one process exiting the critical section and a waiting process entering
- Note: which of these is (more) important depends upon the application domain, and in particular how often critical section access is required

Comparison

	Central Server	Ring	Ricart-Agrawala	Maekawa
Enter	2	0-N	2(N-1)	$2\sqrt{N}$
Client delay	round trip	0-N	round trip	round trip
Synchronization	round trip	1-(N-1)	1	round trip
Exit	0	0	0-(N-1)	$\sqrt{N}$

Central Server Algorithm

- Entering the critical section:
  - requires two messages, the request and the reply — even when no other process currently occupies it
- The client-delay is the time taken for this round-trip
- Exiting the critical section:
  - requires only the sending of the “release” message
  - Incurs no delay for the client, assuming asynchronous message passing.
- The synchronisation-delay is also a round-trip time, the time taken for the “release” message to be sent from client to server and the time taken for the server to send the “grant” message to the next process in the queue.

Ring-based Algorithm

- Entering the critical section:
  - Requires between 0 and *N* messages
- Delay, these messages are serialized so the delay is between 0 and *N*
- Exiting the critical section:
  - Simply requires that the holding process sends the token forward through the ring
- The synchronisation-delay is between 1 and *N*-1 messages

Ricart and Agrawala

- Entering the critical section:
  - This requires 2(*N* - 1) messages, assuming that multicast is implemented simply as duplicated message, it requires *N*-1 requests and *N*-1 replies.
- Delay: Since these messages are sent and received concurrently the time taken is comparable to the round-trip time of the previous two algorithms
  - unless there is contention for bandwidth
- Exiting the critical section:
  - Zero if no other process has requested entry
  - Must send up to *N*-1 responses to queued requests, but again if this is asynchronous there is no waiting for a reply
- The synchronisation-delay is only one message, the holder simply responds to the queued request

Maekawa's Voting algorithm

- Entering the critical section:
  - This requires 2 × sqrt(N) messages
  - As before though, the delay is comparable to a round-trip time
- Exiting the critical section:
  - This requires sqrt(N) messages
  - The delay though is comparable to a single message
  - The total for entry/exit is thus 3 × sqrt(N) which compares favorably to Ricart and Agrawala's total of 2(N − 1) where $N > 4$
- The synchronisation-delay is a round-trip time as it requires the holding process to multi-cast to its voting set the "release" message and then intersecting processes must send a permission granted message to the requesting process

Further Considerations

- The ring-based algorithm continuously consumes bandwidth as the token is passed around the ring even when no process requires entry
- Ricart and Agrawala — the process that last used the critical section can simply re-use it if no other requests have been received in the meantime

Fault Tolerance

- None of the algorithms described above tolerate loss of messages
    - The token based algorithms lose the token if such a message is lost meaning no further accesses will be possible
    - Ricart and Agrawala’s method will mean that the requesting process will indefinitely wait for (N - 1) “permission granted” messages that will never come because one or more of them have been lost
    - Maekawa’s algorithm cannot tolerate message loss without it affecting the system, but parts of the system may be able to proceed unhindered

Process Crashes

- What happens when a process crashes?
- Central server — provided the process which crashes is not the central server, does not hold the token and has not requested the token, everything else may proceed unhindered
- Ring-based algorithm — complete meltdown, but we may get through up to *N*-1 critical section accesses in the meantime
- Ricart and Agrawala — complete meltdown, we might get through additional critical section accesses if the failed process has already responded to them. But no subsequent requests will be granted
- Maekawa’s voting algorithm — This can tolerate some process crashes, provided the crashed process is not within the voting set of a process requesting critical section access