Notes on Distributed Mutual Exclusion
15-440, Fall 2012
Carnegie Mellon University
Randal E. Bryant

Reading: Tannenbaum, Sect. 6.3

Goal:

Maintain mutual exclusion among set of n distributed processes. Each process executes loop of form:

```
while true:
      Perform local operations
      Acquire()
      Execute critical section
      Release()
```

Whereas multithreaded systems can use shared memory, we assume that processes can only coordinate message passing.

Terminology: Define a "cycle" as one round of the protocol, where some process acquires the lock, completes its critical section and then releases it.

Requirements:

1. Safety. At most one process holds the lock at a time

2. Fairness. Any process that makes a request must be granted lock
   A. Implies that system must be deadlock-free
   B. Assumes that no process will hold onto a lock indefinitely
   C. Eventual fairness: Waiting process will not be excluded forever
   D. Bounded fairness: Waiting process will get lock within some bounded number of cycles (typically n)

Other possible goals
   1. Low message overhead
   2. No bottlenecks
   3. Tolerate out-of-order messages
   4. Allow processes to join protocol or to drop out
   5. Tolerate failed processes
   6. Tolerate dropped messages

For today, we will only consider goals 1-3. I.e., assume:

* Total number of processes is fixed at n
* No process fails or misbehaves
* Communication never fails, but messages may be delivered out of order.

Scheme 1: Centralized Mutex Server

Assume there is a single server that acts as a lock manager. It maintains queue Q containing lock requests that have not yet been granted.

Operation on process i

```
Acquire:
    Send (Request, i) to manager
    Wait for reply

Release:
    Send (Release) to manager
```

Operation at server

```
while true:
      m = Receive()
      If m == (Request, i):
         if empty(Q):
            Send (Grant) to i
         else:
            Add i to Q
      If m == (Release) && !empty(Q):
         Remove ID j from Q
         Send (Grant) to j
```

Correctness:

* Clearly safe
* Fairness depends on queuing policy. E.g., if always gave priority to lowest process ID, then processes 1 & 2 could keep making requests & thereby exclude process 3. If use round-robin, or FIFO policy, then would guarantee response within n cycles.

Performance:

* 3 messages per cycle (1 request, 1 grant, 1 release)
* Lock server creates bottleneck

Ricart & Agrawala's algorithm (1981)

Relies on Lamport totally ordered clocks, having the following properties:

1. For any events e, e' such that e --> e' (causality ordering), T(e) < T(e')

2. For any distinct events e, e', T(e) != T(e').

Notation: Ni = {1, 2, ..., i-1, i+1, ..., n} (n is the number of processes)

General idea:

When want to enter C.S., node i sends time-stamped request to all other nodes. These other nodes reply (eventually). When i receives n-1 replies, then can enter C.S.

Trick: Node j having earlier request doesn't reply to i until after it has completed its C.S.

Message types:
    (Request, i, T): Process i requests lock with timestamp T
    (Reply, j): Process j responds to some request for lock

For each node i, maintain following values:

    Ti():
    Function that returns value of local Lamport clock

    should_defer: Boolean
    Set when process i should defer replies to requests

    Tr:
    Time stamp of pending local request

    R: Subset of Ni
    Set of processes from which have received reply

    D: Subset of Ni
    Set of processes for which i has deferred the reply to their requests

    lock(), unlock(): A local mutex lock, to keep the two threads from interfering with each other

Process i consists of two threads. One servicing the application, and one monitoring the network.

Application thread:

```
            Request()             // Request global mutex
            Wait for Notification // Wait until notified by network thread
            Critical Section      // Operate in exclusive mode
            Release()             // Release mutex
```

Application Functions:

```
Request():
        lock()                // Don't want app & network functions to step on each oth
er
        Tr = Ti()             // Get time stamp
        R = {}
        D = {}
        should_defer = true
        Send (Request, i, Tr) to each j in Ni
        unlock()

Release():
        lock()
        should_defer = false
        Send (Reply, i) to each j in D
        unlock()
```

Network Functions:

```
while true:
      m = Receive()
      lock()
      if m == (Request, j, T):
        if should_defer && Tr < T:
             D = D U {j}   // Defer response to j
        else
             Send (Reply, i) to j
      else if m == (Reply, j):
        R = R U {j}
        if R == Ni
             Notify application
      unlock()
```

Performance:

Define a "cycle" to be a complete round of the protocol with one process i entering its critical section and then exiting.

Each cycle involves 2(n-1) messages:

n-1 requests by i
n-1 replies to i

Correctness:

Mutual exclusion: Cannot have two nodes in their critical sections at the same time

Look at nodes A & B. Suppose both are allowed to be in their critical sections at same time.

* A must have sent message (Request, A, Ta) & gotten reply (Reply, A).
* B must have sent message (Request, B, Tb) & gotten reply (Reply, B).

Case 1: One received request before other sent request.

> E.g., B received (Request, A, Ta) before sending (Request, B, Tb). Then would have Ta < Tb. A would not have replied until after leaving its C.S.

Case 2: Both sent requests before receiving others request.

> But still, Ta & Tb must be ordered. Suppose Ta < Tb. Then A would not sent reply to B until after leaving its C.S.

Deadlock Free: Cannot have cycle where each node waiting for some other

Consider two-node case: Nodes A & B are causing each other to deadlock. This would result if A deferred reply to B & B deferred reply to A, but this would require both Ta < Tb & Tb < Ta.

For general case, would have set of nodes A, B, C, ..., Z, such that A is holding deferred reply to B, B to C, ... Y to Z, and Z to A. This would require Ta < Tb < ... < Tz < Ta, which is not possible.

Starvation Free: If node makes request, it will be granted eventually

Claim: If node A makes a request with time stamp Ta, then eventually, all nodes will have their local clocks > Ta.

Justification: From the request onward, every message A sends will have time stamp > Ta. All nodes will update their local clocks upon receiving those messages.

So, eventually, A’s request will have a lower time stamp than any other node’s request, and it will be granted.

Ricart & Agrawala Example

Processes 1, 2, 3. Create totally ordered clocks by having process ID compute timestamp of form T(e) = 10*L(e), where L(e) is a regular Lamport clock.

Initial timestamps-- P1: 421, P2: 112, P3: 143

Action types:
R m: Receive message m
B m: Broadcast message m to all other processes
S m to j: Send message m to process j

Process	T1	T2	T3	Action
	421	112	143	
3			153	B (Request, 3, 153)
2		162		R (Request, 3, 153)
1	431			R (Request, 3, 153)
1	441			S (Reply, 1) to 3
2		172		S (Reply, 2) to 3
3			453	R (Reply, 1)
3			463	R (Reply, 2)
3			473	Enter critical section
1	451			B (Request, 1, 451)
2		182		B (Request, 2, 182)
3			483	R (Request, 1, 451)
3			493	R (Request, 2, 182)
1	461			R (Request, 2, 182)
2		462		R (Request, 1, 451) # 2 has D = {1}
1	471			S (Reply, 1) to 2 # 2 has higher priority
2		482		R (Reply, 1)
3			503	S (Reply, 3) to 1 # Release lock
3			513	S (Reply, 3) to 2
1	511			R (Reply, 3) # 1 has R = {2}
2		522		R (Reply, 3) # 2 has R = {}
2		532		Enter critical section
2		542		S (Reply, 2) to 1 # Release lock
1	551			R (Reply, 2) # 1 has R = {}
1	561			Enter critical section

...

Overall flow: P1 and P2 compete for lock after it is released by P3. P1’s request has timestamp 451, while P2’s request has timestamp 182. P2 defers reply to P1, but P1 replies to P2 immediately. This allows P2 to proceed ahead of P1.

Lamport's Distributed Mutual Exclusion (1978)

Also relies on Lamport totally ordered clocks, having the following properties:

1. For any events e, e' such that e --> e' (causality ordering), T(e) < T(e')

2. For any distinct events e, e', T(e) != T(e').

More complex than R&A:

* 3 rounds of messages.
  - Send Reply message before entering C.S.
  - Send Release message after enterining C.S.
* Each node must maintain local priority queue, orderd by time stamp.

Interesting demonstration of maintaining replica of data any all locations.

Initial version (1978) assumed messages received in same order as sent ("FIFO ordering"). Our version doesn't require this assumption. Only assumes that any message that is sent is eventually received, and that messages are never corrupted.

Message types:
    (Request, i, T): Process i requests lock with timestamp T
    (Reply, j): Process j responds to some request for lock
    (Release): Release lock

For each node i, maintain following values:

    Ti():
    Function that returns value of local Lamport clock

    waiting: Boolean
    Set when process i wants lock

    Q:
    Priority queue with entries of form (j, T), indicating that process j has a request with timestamp T. Ordered so that entry with lowest timestamp at head.

    Tr:
    Time stamp of pending local request

    R: Subset of Ni
    Set of processes from which i has received reply for its request

    D: Subset of Ni
    Set of processes for which i has deferred the reply to their requests

    lock(), unlock():
    A local mutex lock to synchronize the two threads.

Process i consists of two threads. One servicing the application, and one monitoring the network.

Application thread:

```
            Request()             // Request global mutex
            Wait for Notification // Wait until notified by network thread
            Critical Section      // Operate in exclusive mode
            Release()             // Release mutex
```

Application Functions:

```
Request():
        lock()
        Tr = Ti()             // Get time stamp
        R = {}
        D = {}
        Send (Request, i, Tr) to each j in Ni
        Add (i, Tr) to Q
        waiting = true
        unlock()

Release():
        lock()
        Send (Release) to each j in Ni
        Pop top element from Q
        unlock()
```

```
Network Function

while true:
      m = Receive()
      lock()
      if m ==(Request, j, T):
        Add (j, T) to Q
        if waiting && j !in R && Tr < T:
             D = D U {j}   // Defer response to j
        else
             Send (Reply, i) to j
      else if m == (Reply, j):
        R = R U {j}
        if j in D:
            D = D - {j}
            Send (Reply, i) to j
        Check()
      else if m == (Release)
        Pop top element from Q
        Check()
      unlock()

Check():   // Check to see if i is now enabled
        if R == Ni && (i, Tr) at front of queue:
             waiting = false
             Notify application
```

Why does Lamport’s algorithm work?

Key idea:

When process x generates request with time stamp Tx, and it has received replies from all y in Nx, then its Q contains all requests with time stamps <= Tx.

Expressed as follows:

Rule: If x receives message (Reply, y), this indicates that one of the following must hold:

1. y does not have a pending event with time stamp Ty < Tx, or
2. x already has an entry of the form (y, Ty) in Q.

Let’s see how this rule gets implemented. When node i receives (Request, j, T), it does either an "immediate" reply or a "deferred" reply.

IMMEDIATE REPLY. Happens when any of the following conditions hold:

A. !waiting
B. j in R:
C. Tr > T

This will cause j to receive the message (Reply, i). Letting x = j and y = i, we can categorize the three cases as follows:

A. !waiting
   Node i does not want access to critical section
   Rule 1 applies: i does not have a pending event

B. j in R:
   Node j already replied to i’s request
   Rule 2 applies: j has an entry (i, Tr).

C. Tr > T
   Node j’s request has an earlier timestamp.
   Rule 1 applies: i has a pending event, but its time stamp is later than Tr.

DEFERRED REPLY. Occurs after node i receives (Reply, j). That reply is an acknowledgement that j has received i’s request, and so Rule 2 applies.

The deferred reply is the key trick for dealing with out of order messages. By holding back its reply, i will not let j "jump the gun", acting on its own request even though i has an earlier request.

Performance issues:

Define a "cycle" to be a complete round of the protocol with one process i entering its critical section and then exiting.

We can see this cycle would involve 3(n-1) messages as follows:

1. Process i sending n-1 request messages
2. Process i receiving n-1 reply messages
3. Process i sending n-1 release messages.

Alternative organization: Token ring

Idea:

Number processes 0, 1, ..., n-1.

Define next(i) = i + 1 mod n

Processes are logically connected in ring, so that process i can send a message to next(i).

Run two threads for each process, one to service application and one to manage network connection.

Each process i maintains two local Boolean variables:

havetoken:
Initialized to true for process 0 and to false for all others.

waiting:
For application thread to communicate network thread.

Would also need mutex to synchronize changes to these variables, but we will omit these details.

Application functions for process i:

```
Request():
        if havetoken:
           Notify application
        else
           waiting = true

Release():
        havetoken = false
        Send (OK) to next(i)
```

Network functions for process i:

```
        // Starting up
        if havetoken:    // True only for process 0
              Send (OK) to next(i)
              havetoken = false
        // Regular operation
        while true:
              When receive (OK):
                   if waiting:
                      havetoken = true
                      Notify application
                   else
                      Send (OK) to next(i)
```

Correctness:

* Clearly safe: Only one process can hold token
* Fairness: Will pass around ring at most once before getting access.

Performance:

Each cycle requires between 0 & n-1 messages
Latency of protocol between 0 & n-1

Final observations

1. Lamport algorithm demonstrates how distributed processes can maintain consistent replicas of a data structure (the priority queue).

2. Lamport & Ricart & Agrawala’s algorithms demonstrate utility of logical clocks.

3. Centralized & ring based algorithms much lower message counts

4. None of these algorithms can tolerate failed processes or dropped messages.