Notes on Distributed Mutual Exclusion
15-440, Fall 2011
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)
         Remove ID j from Q (block if Q empty)
         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

```
Lamport's Distributed Mutual Exclusion

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)

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

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

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

```
Network Function

while true:
      m = Receive()
      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()

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 i has received replies from all j in Nj, then its Q contains all requests with time stamps <= Tr.

Expressed as follows:

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

1. j does not have a pending event with time stamp T < Tr, or
2. i already has (j, T) in Q.

How do we know this is true?

Consider events

- a1. i sends request (Request, i, Ta). Time stamp Ta1 = Ta
- a2. j receives request (Request, i, Ta). Time stamp Ta2
- a3. j sends reply (Reply, j). Time stamp Ta3
- a4. i receives reply (Reply, j). Time stamp Ta4

Suppose there is another set of events:

- b1. j sends request (Request, j, Tb). Time stamp Tb1 = Tb
- b2. i receives request (Request, j, Tb). Time stamp Tb2
- b3. i sends reply (Reply, i). Time stamp Tb3
- b4. j receives reply (Reply, i). Time stamp Tb4.

Each of these sequences has causal ordering constraints

Ta = Ta1 < Ta2 < Ta3 < Ta4

Tb = Tb1 < Tb2 < Tb3 < Tb4

Violating the rule above would require a scenario where Tb < Ta but Ta4 < Tb2.

Assuming Tb < Ta, the code for Request() at i, would have put j in D at time Ta2, and it would not have sent (Reply, j) to i until after Tb4. So we must have Tb4 < Ta3. We can combine this with the other orderings to get Tb2 < Tb4 < Ta3 < Ta4, and so our potential error cannot occur.

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.

Ricart & Agrawala's algorithm

This algorithm is a refinement to Lamport's. The main idea is to combine the role of the Reply and the Release messages. That is, process j will not send (Reply, j) to i until after it has completed any local event with time stamp T < Tr.

We can dispense with much of the machinery of Lamport's algorithm, including the priority Q.

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

    waiting: Boolean
    Set when process i wants lock

    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

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

Release():
        Send (Reply, j) to each j in D
```

Network Functions:

```
while true:
      m = Receive()
      if m == (Request, j, T):
        if waiting && 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
             waiting = false
             Notify application
```

Performance:

Each cycle involves 2(n-1) messages:

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

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.

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.