Distributed Mutual Exclusion

Ch 10 [BenA 06]

Distributed System
Distributed Critical Section
Ricart-Agrawala
Token Passing Ricart-Agrawala
Token Passing Neilsen-Mizuno
(Generic) Distributed System

- Nodes have processes
- Communication channels between nodes
  - Each node connected to every other node
    - Two-way channel
  - Reliable communication channels
    - Provided by network layer below
    - Messages are not lost
    - Messages processed concurrently with other computations (e.g., critical sections)
  - Nodes do not fail
- Requirements reduced later on
  - courses on distributed systems topics

Unrealistic assumptions? Not really…
(Generic) Distributed System

- Processes (nodes) communicate with (asymmetric) messages
  - Message arrival order is not specified
  - Transmission times are arbitrary, but finite
  - Message (header) does not include send/receiver id
  - Receiver does not know who sent the message
    - Unless sender id is in the message itself

```
node 5

integer k ← 20
send(request, 3, k, 30)

node 3

integer m, n
receive(request, m, n)
```
Distributed Processes

- Sender does not block
- Receiver blocks (suspended wait) until message of the proper type is received
- Atomicity problems in each node is not considered here
  - Solved with locking, semaphores, monitors, …
- Message receiving and subsequent actions are considered to be atomic actions
  - Atomicity within each system considered solved
Distributed Critical Section Problem

- Processes within one node
  - Problem solved before
- Processes in different nodes
  - More complex
- State
  - Control pointer (CP, PC, program counter)
  - Local and shared variable values
  - Messages
    - Messages, that have been sent
    - Messages, that have been received
    - Messages, that are on the way
      - Arbitrary time, but finite!

Where are these?
Two Approaches

- Ask everybody for permission, if it is my turn now
  - Lots of questions/answers
- I’ll wait until I get the token, then it is my turn
  - Pass the token to next one (which one?)
  - Wait until I get the token
  - Token (turn) goes around all the time
    - Moves only when needed?
- Both approaches have advantages/disadvantages
  - Who is “everybody”? How do I know them?
  - What if someone does not talk to me?
  - What if node/network breaks down?
  - What if token is lost?

Do not worry now about the token getting lost …
Ricart-Agrawala for Distributed Mutex

- Distributed Mutex, 1981 (Lamport, 1978)
- Modification of Bakery algorithm with ticket numbers
- Idea
  - Must know all other processes/nodes competing for CS
  - Choose own ticket number, “larger than previous”
  - Send it to everybody else
  - Wait until permission from everybody else
    - Exactly one will always get permission from everybody else?
    - All others will wait
  - Do your CS
  - Give CS permission to everybody else who was waiting for you
### Algorithm 10.1: Ricart-Agrawala algorithm (outline)

```plaintext
integer myNum ← 0
set of node IDs deferred ← empty set
```

<table>
<thead>
<tr>
<th>main</th>
<th>application process, needs distr mutex</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1:</td>
<td>non-critical section</td>
</tr>
<tr>
<td>p2:</td>
<td>myNum ← chooseNumber</td>
</tr>
<tr>
<td>p3:</td>
<td>for all other nodes N</td>
</tr>
<tr>
<td>p4:</td>
<td>send(request, N, myID, myNum)</td>
</tr>
<tr>
<td>p5:</td>
<td>await reply's from all other nodes</td>
</tr>
<tr>
<td>p6:</td>
<td>critical section</td>
</tr>
<tr>
<td>p7:</td>
<td>for all nodes N in deferred</td>
</tr>
<tr>
<td>p8:</td>
<td>remove N from deferred</td>
</tr>
<tr>
<td>p9:</td>
<td>send(reply, N, myID)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>receive</th>
<th>server process, runs concurrently all the time</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer source, reqNum</td>
<td></td>
</tr>
<tr>
<td>p10:</td>
<td>receive(request, source, reqNum)</td>
</tr>
<tr>
<td>p11:</td>
<td>if reqNum &lt; myNum</td>
</tr>
<tr>
<td>p12:</td>
<td>send(reply, source, myID)</td>
</tr>
<tr>
<td>p13:</td>
<td>else add source to deferred</td>
</tr>
</tbody>
</table>
```

- Local mutex control?
- not trivial!
- Each one answers only when it is safe. Reply needs no content.
- all those waiting for my permission
- most recent myNum
- make these wait by not sending reply
Ricart-Agrawala Example

- 3 processes, each trying to enter CS concurrently
  - No status information needed on who had CS last
Ricart-Agrawala Example (contd)

- Receive process runs at each node
  - What if Aaron’s \textit{receive} completes 1st? Last? Becky’s? not yet?

if reqNum < myNum  
send(reply,source,myID)  
else add source to deferred

Distributed virtual queue:

deferred, can enter CS after me

I got reply from everybody, I can enter CS

\begin{itemize}
  \item Distributed virtual queue:
\end{itemize}
Ricart-Agrawala Example (contd)

- Becky executes CS, and then sends deferred replies to Aaron & Chloe
- Aaron has now replies from everybody, and it can enter CS
- What if Becky now selects ticket number 8, and requests CS?
  - Aaron’s and Chloe’s receive will both reply immediately? Ouch!

```java
if reqNum < myNum
    send(reply, source, myID)
else add source to deferred
```

Problem: Becky’s ticket number 8 is too small
(Becky should not be able to select so small number)
How to select ticket numbers

• Select always larger one than you have seen before
  – Larger than your previous \textit{myNum}
  – Larger than any \textit{requestedNum} that you have seen
    • They all came before you, and you should not try to get ahead of them

• What if equal ticket numbers?
  – Fixed priority, based on node/process id numbers
  – Used only with equal ticket numbers to avoid deadlock
    • Just like in Bakery algorithm
Quiescent Nodes

- Nodes that do not try to enter CS (but they could)
  - They are still listed in “all other nodes”
  - Problem with initial value of myNum
  - Initial value zero?

- Initial value N > 0 ; tickets numbers eventually will reach it

- Cure: receive checks for tickets numbers only if main wants CS
Algorithm 10.2: Ricart-Agrawala algorithm

integer myNum ← 0
set of node IDs deferred ← empty set
integer highestNum ← 0

Main

loop forever
p1: non-critical section
p2: requestCS ← true
p3: myNum ← highestNum + 1
p4: for all other nodes N
p5: send(request, N, myID, myNum)

p6: await reply’s from all other nodes
p7: critical section
p8: requestCS ← false
p9: for all nodes N in deferred
p10: remove N from deferred
p11: send(reply, N, myID)

• Keep track of highest number seen
• What if one process asks for CS all the time?
• Same myNum OK?

(Receive on next slide)
Algorithm 10.2: Ricart-Agrawala algorithm (continued)

Receive

<table>
<thead>
<tr>
<th>integer source, requestedNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
</tr>
<tr>
<td>p1: receive(request, source, requestedNum)</td>
</tr>
<tr>
<td>p2: highestNum ← max(highestNum, requestedNum)</td>
</tr>
<tr>
<td>p3: if not requestCS or requestedNum ≪ myNum</td>
</tr>
<tr>
<td>p4: send(reply, source, myID)</td>
</tr>
<tr>
<td>p5: else add source to deferred</td>
</tr>
</tbody>
</table>

- Mutex between main & receive?
  - Exact mutex boundaries?
- What to do when myNum overflows?
  - Restart everybody? When? How?
  - Fairness is not the problem, mutex is
- Correctness proofs
  - Mutex? No deadlock? No starvation?

Discussion B

original article

http://www.cc.gatech.edu/classe/AY2002/cs6210_fall/papers/MutualExForNetwork.pdf
Token Based Algorithms

- Problems with permission based algorithms
  - Need permission from everybody (very many?)
  - Inactive participants (those not wanting in CS) slow you down
    - Need reply from all of them!
    - Lots of synchronization even if only one tries to get into CS
    - Lots of communication (many messages)

- Token based algorithms
  - Have token, that is enough
    - No synchronization with everybody else needed
  - Get token, send token is simple
    - Communicate only with a few (fewer) nodes
    - Scalable?
  - Mutex is trivial, how about deadlock and starvation?
Ricart-Agrawala ideas

- Send token to next one only when I know that someone wants it
  - o/w keep token until needed
- Keep local *requested* array for best knowledge for the most recent CS request times
  - Update this based on received CS request messages
- Keep *granted* array, that has precise knowledge when each node actually was last granted CS
  - Update it only when CS granted
  - Pass it with token to next node
    - Only this *granted* array (with token) is exactly correct!
    - Other nodes have (slightly) old *granted* array
Algorithm 10.3: Ricart-Agrawala token-passing algorithm

```java
boolean haveToken ← true in node 0, false in others
integer array[NODES] requested ← [0, . . . , 0]
integer array[NODES] granted ← [0, . . . , 0]
integer myNum ← 0
boolean inCS ← false

sendToken
if exists N such that requested[N] > granted[N]
  for some such N
    send(token, N, granted)
    haveToken ← false
```

**Receive**

server process, runs all the time

```java
integer source, reqNum
loop forever
  receive(request, source, reqNum)
  requested[source] ← max(requested[source], reqNum)
  if haveToken and not inCS
    sendToken
    Give also most recent granted[]

  if no one else wants token, I will keep it
```
Algorithm 10.3: Ricart-Agrawala token-passing algorithm (continued)

Main application process, needs distr mutex

loop forever
  non-critical section
  if not haveToken
    myNum ← myNum + 1
    for all other nodes N
      send(request, N, myID, myNum)
  haveToken ← true
  inCS ← true
  critical section
  granted[myID] ← myNum
  inCS ← false
  sendToken

If I have token, no delays.

Request token from everybody

Very many messages?

Wait until token received

Just one very large message?

- Mutex?
- No deadlock?
- No starvation?
  - “some” in sendToken?
- Scalable?
- Overflows?

Send granted also.

Only if someone wants it!
Algorithm 10.3: Ricart-Agrawala

Main application process

| loop forever |
| non-critical section |
| if not haveToken |
| myNum ← myNum + 1 |
| for all other nodes N |
| send(request, N, myID, myNum) |
| receive(token, granted) |
| haveToken ← true |
| inCS ← true |
| critical section |
| granted[myID] ← myNum |
| inCS ← false |
| sendToken |

if I have token, no delays.

Only if someone wants it! Send granted also.

Very many messages?

- Can Chloe be 3rd time in CS?
- Who wants CS now?
- If Chloe has token, and is in non-CS, what happens next?
- If Chloe has token and is in CS, what happens next?
- Why is Chloe's own requested[i] zero?
- Could Becky have kept the token since last use?
Neilsen-Mizuno

Token Based Algorithm

- Rigart-Agrawala: token carries queue of waiting processes
  - Token can be very large, which may be problematic
- Neilsen-Mizuno: virtual tree structure within the nodes implements the queue
  - Algorithm utilizes virtual spanning tree of nodes
    - Spanning tree: all nodes linked as a tree, no cycles
  - Simple token indicates “turn” for critical section
  - Parent link points to the direction of last in line for CS
    - Parent == 0: node may have token and is last in line for CS
  - Deferred link points to next in line for CS

Chloe has token, Aaron is waiting for it
Neilsen-Mizuno Example

- Fully connected nodes
- Chloe is in CS
- No one waits for CS
Neilsen-Mizuno Example (contd)

- Chloe has token, nobody waits for it

- Aaron requests CS
  - Sends msg=(req, Aaron, Aaron) on parent link
  - Removes himself from parent spanning tree

- Becky receives msg, and forwards the request “upward”
  - Sends msg=(req, Becky, Aaron) to Chloe
  - Moves to new parent spanning tree, points to Aaron
  - Aaron is now last to request CS
Neilsen-Mizuno Example (contd)

Chloe receives msg (req, Becky, Aaron)
- Chloe in CS, sets deferred field to Aaron and sets parent field to Becky
  - Chloe was (also) last in line for CS

- When Chloe completes CS, she will pass token to Aaron
  - Token transferred directly to the next process in line for critical section (if any)
    - Just token is passed, no big array with it
Neilsen-Mizuno Example (contd)

- Chloe still has CS, Evan wants CS
  - Sends (req, Evan, Evan) to Danielle
  - Danielle sends (req, Danielle, Evan) to Chloe
  - Chloe sends (req, Chloe, Evan) to Becky
  - Becky sends (req, Becky, Evan) to Aaron
  - Aaron makes a deferred link to Evan
Neilsen-Mizuno Example (contd)

- Chloe completes CS, passes token to Aaron

- Aaron completes CS, passes token to Evan

- Evan completes CS, keeps token
Algorithm 10.4: Neilsen-Mizuno token-passing algorithm

integer parent ← (initialized to form a tree)
external deferred ← 0

boolean holding ← true in the root, false in others

Main

loop forever

1. non-critical section

2. if not holding

3. send(request, parent, myID, myID)

4. parent ← 0

5. receive(token)

6. holding ← false

7. critical section

8. if deferred ≠ 0

9. send(token, deferred)

10. deferred ← 0

11. else holding ← true
## Algorithm 10.4: Neilsen-Mizuno token-passing algorithm

<table>
<thead>
<tr>
<th>Receive</th>
<th>(runs concurrently with main, mutex problems solved...)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>integer source, originator</td>
</tr>
<tr>
<td></td>
<td>loop forever</td>
</tr>
<tr>
<td>p12:</td>
<td>receive(request, source, originator)</td>
</tr>
<tr>
<td>p13:</td>
<td>if parent = 0 last in queue</td>
</tr>
<tr>
<td>p14:</td>
<td>if holding have token, not in CS</td>
</tr>
<tr>
<td>p15:</td>
<td>send(token, originator)</td>
</tr>
<tr>
<td>p16:</td>
<td>holding ← false</td>
</tr>
<tr>
<td>p17:</td>
<td>else deferred ← originator place new req last in queue</td>
</tr>
<tr>
<td>p18:</td>
<td>else send(request, parent, myID, originator) forward request</td>
</tr>
<tr>
<td>p19:</td>
<td>parent ← source update direction for last request</td>
</tr>
</tbody>
</table>
Ricart-Agrawala vs. Neilsen-Mizuno

- Number of messages needed
- Size of messages
- Size of data structures in each node
- Behaviour with heavy load
  - Many need CS at the same time
- Behaviour with light load
  - Requests for CS do not come often
  - Usually only one process requests CS at a time
Other Distributed Mutex Algorithms

• Other token-based algorithms
  – Token ring: token moves all the time
  – Lots of token traffic even when no CS requests

• Centralized server
  – Simple, not very many messages
  – Not scalable, may become bottleneck

• Give up unrealistic assumptions
  – Nodes may fail
  – Messages may get lost, token may get lost

• See other courses

Courses on distributed systems topics (hajautetut järjestelmät)
Summary

- Distributed critical section is hard, avoid it
  - Use centralized solutions if possible?
- Permission based solutions
  - Ricart-Agrawala – ask everyone
- Token based solutions
  - Ricart-Agrawala – centralized state in granted[
  - Neilsen-Mizuno – queue kept in spanning tree
- There are other algorithms
- How do they scale up?