Distributed Mutual Exclusion

Ch 10 [BenA 06]

Distributed System
Distributed Critical Section
Ricart-Agrawala
Token Passing Ricart-Agrawala
Token Passing Neilsen-Mizuno

1.12.2009 Copyright Teemu Kerola 2009

(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

1.12.2009 Copyright Teemu Kerola 2009
(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

![Diagram]

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?

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

Do not worry now about the token getting lost …
Algorithm 10.1: Ricart-Agrawala algorithm (outline)

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

main application process, needs distr mutex

{local mutex control?}

\begin{enumerate}
\item non-critical section
\item myNum ← chooseNumber
\item for all other nodes N
\item send(request, N, myID, myNum)
\item await reply's from all other nodes
\item critical section
\item for all nodes N in deferred
\item remove N from deferred
\item send(reply, N, myID)
\end{enumerate}

receive server process, runs concurrently all the time

\begin{enumerate}
\item integer source, reqNum
\item receive(request, source, reqNum)
\item if reqNum < myNum
\item send(reply,source,myID)
\item else add source to deferred
\end{enumerate}

Each one answers only when it is safe.
Reply needs no content.
All those waiting for my permission

- Local mutex control?
- Not trivial!

Ricart-Agrawala Example

- 3 processes, each trying to enter CS concurrently
  - No status information needed on who had CS last

\begin{figure}
\centering
\includegraphics[width=\textwidth]{ricart-agrawala_example}
\end{figure}
Ricart-Agrawala Example (contd)

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

<table>
<thead>
<tr>
<th>myNum</th>
<th>deferred, can enter CS after me</th>
</tr>
</thead>
</table>

### Ricart-Agrawala Example (contd)

- Distributed virtual queue:

```
I got reply from everybody, I can enter CS
```

- 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!

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

```html
Ricart-Agrawala Example (contd)
```

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

```html
Ricart-Agrawala Example (contd)
```
How to select ticket numbers

- Select always larger one than you have seen before
  - Larger than your previous myNum
  - Larger than any 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?
    - if reqNum < myNum
      - send(reply, source, myID)
    - else add source to deferred

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

- Cure: receive checks for tickets numbers only if main wants CS
Concurrent Programming (RIO) 1.12.2009

Lecture 10: Distributed Mutual Exclusion

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
integer source, requestedNum
loop forever
p1: receive(request, source, requestedNum)
p2: highestNum ← max(highestNum, requestedNum)
p3: if not requestCS or requestedNum ≤ myNum
p4: send(reply, source, myID)
p5: else add source to deferred

• 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?

1.12.2009 Copyright Teemu Kerola 2009
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] ← local data in node
integer array[NODES] granted ← [0,...,0] ← distributed global data
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

else
  if no one else wants token, I will keep it

Receive

server process, runs all the time

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 requested[source] > granted[source]
    requested[source] ← requested[source]
```

1.12.2009 Copyright Teemu Kerola 2009
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)
    receive(token, granted)
    haveToken ← true
    if I have token, no delays.
  inCS ← true
  critical section
  granted[myID] ← myNum
  inCS ← false
  sendToken ← true
  Only if someone wants it!
  Send granted also.

Discussion C

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

Application process, needs distr mutex

If I have token, no delays.
Request token from everybody
Very many messages?

Wait until token received
Just one very large message?

Update one field

Only if someone wants it!
Send granted also.

Discussion C

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
  
  Chloe
  Becky
  Danielle
  Evan

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

  Aaron
  Becky
  Chloe
  Danielle
  Evan

- 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
  Becky
  Chloe
  Danielle
  Evan

  Aaron
  Becky
  Chloe
  Danielle
  Evan

  Aaron
  Becky
  Chloe
  Danielle
  Evan

  Chloe
  Becky
  Chloe
  Danielle
  Evan

- Chloe receives msg (req, Becky, Aaron)
  - Chloe in CS, sets deferred field to Aaron
  - 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

```
Main

loop forever
  p1: non-critical section
  p2: if not holding
    p3: send(request, parent, myID, myID)
    p4: parent ← 0
    p5: receive(token)
    p6: holding ← false
    p7: critical section
  p8: if deferred ≠ 0
    p9: send(token, deferred)
    p10: deferred ← 0
    p11: else holding ← true
```

Discussion D

Algorithm 10.4: Neilsen-Mizuno token-passing algorithm

```
Receive (runs concurrently with main, mutex problems solved...)

loop forever
  p12: receive(request, source, originator)
  p13: if parent = 0
    p14: if holding
      p15: send(token, originator)
    p16: holding ← false
  p17: else deferred ← originator
    p18: else send(request, parent, myID, originator)
    p19: parent ← source
```

Discussion D
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?