Algorithms for Distributed Mutual Exclusion

Dr Vladimir Z. Tasic
Term 2 2020
IN THE NEXT 4 LECTURES...

- The context: distributed systems using message passing
- Some common concurrency problems, classical algorithms and their strengths/weaknesses
  - Modeling (abstraction) of distributed systems
  - Distributed critical sections (distributed mutual exclusion)
  - Handling inconsistent information in case of failures
  - Determining termination and global property snapshots
- Additional concurrency paradigms, e.g. the actor model
- Some additional concurrency programming constructs
MAIN TOPICS IN THIS LECTURE...
(PARTLY IN BEN-ARI CHAPTER 10)

• Some “Big Ideas” about concurrency in distributed systems (and wider)
• Ben-Ari’s distributed system model (remember these assumptions!)
• Ricart-Agrawala algorithm for distributed mutual exclusion (distributed critical sections)
• Token-passing algorithms for distributed mutual exclusion - another Ricart-Agrawala algorithm
SOME “BIG IDEAS” ON CONCURRENCY IN DISTRIBUTED SYSTEMS (& WIDER)

From various sources, including personal experience
BIG IDEA 1: KNOW THY CONTEXT

• **What works well in one context** ...
  • ... might fail miserably in another context
  • ... work, but not so well, in another context
  • ... work also (or even better) in another context
• **Understand the** theoretical and practical **intrinsicacies of the context** you are working in - “the devil is in the details”

• Know also solutions from somewhat similar contexts - possible cross-pollination
OUR CONTEXT: DISTRIBUTED SYSTEMS

- Loosely coupled independent computers
  - When no central point of control - decentralised system
- Each computer has local memory
- In almost all cases no shared memory
- Communication by message passing over a communications network
- Possible errors or failures in the communication network or the computers
SOME IMPACTS OF THE DISTRIBUTED SYSTEMS CONTEXT

• In complex, decentralised systems (e.g. on the Internet)
  • message passing has advantages over shared memory
  • asynchronous communication has advantages over synchronous communication
  • immutable data has advantages over mutable data
  • …

• Flexibility is needed because in distributed systems change is frequent and often unpredictable
  • caused by technology or by business aspects
BIG IDEA 2: UNDERSTAND ASSUMPTIONS OF THE USED MODELS

• **Models** abstract unnecessary details to enable focusing on the aspects we care about

• Unfortunately, all models are **simplifications** developed under some **assumptions**

• **Understand assumptions and limitations** of the model that you use

• ... but also of **models that underpin** the systems, languages, libraries, **algorithms**, ... that you use
THE COST OF NEGLECTING THE UNDERLYING ASSUMPTIONS

• If you do something that does not satisfy some underlying assumptions …
• … the result might be irrelevant (but sometime the errors are huge) - unpredictability

• For example …

• Thus: understand assumptions of Ben-Ari’s distributed systems model to reason about the studied algorithms
BIG IDEA 3: LEARN BOTH THEORY AND PRACTICE

- “Experience without theory is blind, but theory without experience is mere intellectual play” Immanuel Kant
- **Theoretical knowledge** and **formal reasoning** are **indispensable** for developing concurrent systems.
- **Practical experience** helps to understand **intricacies of your context**, as well as whether **assumptions** of the used models are **realistic** in your context.
  - *Would you drive a car the safety of which was checked only on mathematical models (and, possibly, computer simulations) without crash-test dummies?*
Concurrent software leverages modern hardware better!

- Availability of multi-core/multi-processor hardware: a system currently running on a single processor might soon need to run on a multi-core/multi-processor computer
- Distribution due to technical and business reasons: a system running on an in-house server might soon need to be running in a cloud environment with required scaling and elasticity
- Modifying sequential software to become concurrent software can be a nightmare!
- For example …
BIG IDEA 5: MASTER SEVERAL (NEW AND OLD) CONCURRENCY PARADIGMS

• “If your only tool is a hammer, then every problem looks like a nail”
  • P.S. Most problems are NOT nails

• Modern concurrent computing is much more than threads and locks (semaphores, monitors)

• **Different concurrency paradigms** are used for different problem types or in different contexts

• **Comeback** of some old computer science ideas - changed context led to new use cases for old ideas
  • E.g., the actor model from 1973 became very popular in 2010s due to cloud computing
DIFFERENT CONCURRENCY MODELS - INTRODUCTORY READING/WATCHING

- **Task for you:** Read the free *Chapter 1 “Introduction”* (hyperlink) from
- **Then, watch the video:**
  - Parleys, “Comparing different concurrency models on the JVM” [video, 53:31], YouTube, 4 Jan. 2016, at: https://www.youtube.com/watch?v=QFB_3uUGzR4
- **Think about this (!) and use it in your Assignment 2**
THE 7 CONCURRENCY MODELS
BY BUTCHER

1. Threads and locks
2. Functional programming
3. Separating identity and state
4. The actor model
5. Communicating Sequential Processes
6. Data parallelism
7. The Lambda Architecture (using map-reduce, streams)
A CONCURRENT PROGRAMMING TOOLBOX
BEN-ARI’S DISTRIBUTED SYSTEMS MODEL

From Chapter 10 in *Ben-Ari’s Textbook*
BEN-ARI’S DISTRIBUTED SYSTEMS
MODEL ASSUMPTIONS (1/3)

- **Node**: physical object (computer, printer, etc.) with unique ID
  - Nodes can be heterogeneous
- **Process**: sequential program, a sequence of actions that produce a result
- Communication within 1 node using shared memory, between nodes only using message passing
- (Assume for now) **No or only limited failures** in nodes so that cooperation between nodes is not impacted by node failures
- **Fully connected topology**: 2-way communication (possibly multi-hop) between each pair of nodes
BEN-ARI’S DISTRIBUTED SYSTEMS MODEL ASSUMPTIONS (2/3)

- Messages delivered **without error** (after retransmissions or corrections by the communications system), but **possibly in different order** from the one in which they were sent
  - E.g. TCP/IP can be used for such communications
- Message travel times are **finite but arbitrary**
- **send**(*MessageType*, *Destination*, *[Parameters]*)  // IDs not sent
- **receive**(*MessageType*, *[Parameters]*)  // note: from any Source
- If needed, Source ID can be a message parameter
Pre- and post-protocol for CS (critical section) are treated atomic, while CS and NCS (non-critical section) need not be.

Receiving and handling of a message is 1 atomic statement and interleaving with other processes on the same node is prevented.

To understand a distributed algorithm, you need to know for each node its state, local data and exchanged messages.

Task for you: Download Ben-Ari’s teaching tool DAJ (URL: https://github.com/motib/daj), read the first 6 pages of its user manual and experiment with Ricart-Agrawala algorithm in DAJ.
PARALLEL VIRTUAL MACHINE (PVM)

• A distributed system implementation providing an abstract view of the underlying network

• Regardless of the actual network configuration, programmer sees a set of nodes and can freely assign processes to nodes

• Architecture of the virtual machine can be changed dynamically by any node, supporting fault-tolerance

• Interoperability: a program can run on node of any type and can exchange messages with nodes of any type
MESSAGE PASSING INTERFACE (MPI) LIBRARY

• **MPI** is standardised library interface for message passing
• OpenMPI (sometimes MPICH) in Linux distributions
• Traditionally: **SPMD (Single Program, Multiple Data)**
  • The same program for all nodes; a copy is loaded onto every node; behaviour can be varied by checking process ID (rank)
• Nowadays: also **MPMD (Multiple Programs, Multiple Data)**
• MPI_Send (basically) non-blocking, while MPI_Recv blocking
• **FYI: A tutorial is at:** [https://computing.llnl.gov/tutorials/mpi/](https://computing.llnl.gov/tutorials/mpi/)  (URL)
DISTRIBUTED MUTUAL EXCLUSION
(DISTRIBUTED CRITICAL SECTIONS)

From Chapter 10 in *Ben-Ari’s Textbook*
THE NEED FOR DISTRIBUTED CRITICAL SECTIONS (DISTRIBUTED. MUTUAL EXCLUSION)

• Task for you: Provide examples in which critical sections with mutual exclusion are needed in a distributed system with no shared memory
RICART-AGRAWALA ALGORITHM - MAIN IDEAS

• Using ticket numbers, similarly to Lamport’s bakery algorithm
• Nodes choose ticket numbers and compare them

• In a distributed system these numbers cannot be compared directly, so they have to be sent in messages

• Node with lowest number can enter CS (critical section)
• Other nodes have to wait until CS is free again
**Algorithm 10.1: Ricart-Agrawala algorithm (outline)**

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

<table>
<thead>
<tr>
<th>Main</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1:</td>
</tr>
<tr>
<td>p2:</td>
</tr>
<tr>
<td>p3:</td>
</tr>
<tr>
<td>p4:</td>
</tr>
<tr>
<td>p5:</td>
</tr>
<tr>
<td>p6:</td>
</tr>
<tr>
<td>p7:</td>
</tr>
<tr>
<td>p8:</td>
</tr>
<tr>
<td>p9:</td>
</tr>
</tbody>
</table>
```
• p12: receiving node agrees that this sending node enters CS
• Node with lowest number will receive replies from all other nodes
• p13: receiving node defers replying so this sending node (with higher ticket number) cannot yet enter CS
RICART-AGRAWALA ALGORITHM - EXAMPLE (1/2)

Fig. 1: after all nodes chose ticket numbers and sent request messages

Fig. 2: after all nodes executed Receive process for all messages; ● in CS
Fig. 3: after 1st in the **virtual queue** completed CS and replied to all deferred nodes.

Fig. 4: after 2nd in the virtual queue completed CS and replied to the deferred node.
• **Virtual queue** does not exist as a data structure, but it is the effect of messages - nodes ordered as if in a queue
  
  • Example: Becky, Aaron, Chloe

• **Many other message interleaving scenarios** possible

• **Equal ticket numbers** handled as in the bakery algorithm:

  \[
  \text{if requestNum} \ll \text{myNum}
  \]

  means:

  \[
  \text{if (requestNum < myNum) or ((requestNum = myNum) and (source < myID))}
  \]
• Chosen ticket numbers must be monotonic, i.e. higher than all other ticket numbers a node knows about

• To each node add variable highestNum and change \( p2 \) to:
  \[ p2: \text{myNum} \leftarrow \text{highestNum} + 1 \]
and add after \( p10 \):
  \[ p10.5: \text{highestNum} \leftarrow \max(\text{highestNum}, \text{requestedNum}) \]

• To handle quiescent (inactive) nodes, Main sets flag resetCS before choosing number and resets after exiting CS

• If resetCS is not set, Receive immediately sends reply
### Algorithm 10.2: Ricart-Agrawala algorithm

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer myNum ← 0</td>
</tr>
<tr>
<td>set of node IDs deferred ← empty set</td>
</tr>
<tr>
<td>integer highestNum ← 0</td>
</tr>
<tr>
<td>boolean requestCS ← false</td>
</tr>
<tr>
<td>Main</td>
</tr>
<tr>
<td>------------------------------------------</td>
</tr>
<tr>
<td>loop forever</td>
</tr>
<tr>
<td>p1: non-critical section</td>
</tr>
<tr>
<td>p2: requestCS ← true</td>
</tr>
<tr>
<td>p3: myNum ← highestNum + 1</td>
</tr>
<tr>
<td>p4: for all other nodes N</td>
</tr>
<tr>
<td>p5: send(request, N, myID, myNum)</td>
</tr>
<tr>
<td>p6: await reply’s from all other nodes</td>
</tr>
<tr>
<td>p7: critical section</td>
</tr>
<tr>
<td>p8: requestCS ← false</td>
</tr>
<tr>
<td>p9: for all nodes N in deferred</td>
</tr>
<tr>
<td>p10: remove N from deferred</td>
</tr>
<tr>
<td>p11: send(reply, N, myID)</td>
</tr>
</tbody>
</table>
### Receive

<table>
<thead>
<tr>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer source, requestedNum</td>
</tr>
<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>
• Theorem 10.1: Mutual exclusion holds
• Proof by contradiction: Assume nodes $i$ and $j$ are in CS
  - Case 1: Node $j$ chose $myNum_j$ after it sent its reply to node $i$
  - Case 2: Node $i$ chose $myNum_i$ after it sent its reply to node $j$

Symmetrical to Case 1

$i$ and $j$ cannot be in CS at the same time, due to $highestNum$
- Case 3: Nodes $i$ and $j$ chose $myNum_i$ and $myNum_j$ before sending reply messages to each other.

$i$ and $j$ cannot be in CS at the same time, due to $\ll$.
RICART-AGRAWALA ALGORITHM - FREEDOM FROM STARVATION

• Theorem 10.2: **Free from starvation** and, consequently, **free from deadlock**

  • Proof: Node \( i \) sets \( \text{requestCS} \), choses \( myNum_i \), sends request messages and \( \text{awaits} \) replies from all other nodes at \( p6 \)

  • Eventually, these messages will arrive and any other process \( j \) trying to enter CS will choose \( myNum_j > myNum_i \)

  • \( \text{aheadOf}(i) \) is set of nodes that at time \( t \) have already chosen \( myNum < myNum_i \), but due to **monotonicity under \( \ll \)** no new process is added and eventually all of them complete CS

  • Then, \( myNum_i \) will be minimal so \( i \) will be able to enter CS
Global variables within nodes are arrays, 1 element per node:

- byte myNum[NPROC];
- byte highestNum[NPROC];
- bool requestCS[NPROC];
- chan deferred[NPROC] = [NPROC] of {byte};

Use of `atomic` to prevent interleaving.

Channels between nodes are many-to-1:

- mtype = {request, reply};
- chan ch[NPROC] = [NPROC] of {mtype, byte, byte};
RICHART-AGRAWALA ALGORITHM - PROMELA FOR Main

proctype Main( byte myID ) {
   do ::
      atomic {
         requestCS[myID] = true;
         myNum[myID] = highestNum[myID] + 1;
      }
   for (J,0,NPROCS−1)
      if :: J != myID –>
         ch[J] ! request, myID, myNum[myID];
      else
         fi
      rof (J);
   for (K,0,NPROCS−2)
      ch[myID] ?? reply, _, _;
      rof (K);
      critical_section ();
      requestCS[myID] = false;
   byte N;
   do
      :: empty(deferred[myID]) –> break;
      :: deferred[myID] ? N –> ch[N] ! reply, 0, 0
      od
   od
}
Ricart-Agrawala Algorithm - PROMELA for Receive

```promela
proctype Receive( byte myID ) {
    byte reqNum, source;
    do ::
        ch[myID] ?? request, source, reqNum;
        highestNum[myID] =
            ((reqNum > highestNum[myID]) ->
            reqNum : highestNum[myID]);

    atomic {
        if :: requestCS[myID] &&
            ( (myNum[myID] < reqNum) ||
            ( (myNum[myID] == reqNum) &&
                (myID < source) ) ) ->
            deferred [myID] ! source
            ch[source] ! reply, 0, 0 fi
    od
}
```
RICART-AGRAWALA ALGORITHM - LIMITATIONS

- **Ever-increasing ticket numbers** (same as in Lamport’s bakery algorithm)
  - This can be a problem in long-running systems

- **Can be inefficient** for large number of nodes
- **Performance not improved** when there is no contention - node wishing to enter CS must exchange messages with all other nodes

*Optional task for you: Read about various distributed mutual exclusion algorithms, e.g. see* [https://www.cs.uic.edu/~ajayk/Chapter9.pdf](https://www.cs.uic.edu/~ajayk/Chapter9.pdf)
TOKEN PASSING ALGORITHMS FOR DISTRIBUTED MUTUAL EXCLUSION

From Chapter 10 in *Ben-Ari’s Textbook*
**TOKEN-PASSING ALGORITHMS**

- **Token** denotes permission to enter CS
- **Only 1 node has token at any 1 time** - mutual exclusion holds
- **Efficiencies**:
  - only 1 message needed to transfer token between nodes
  - node with token can enter CS multiple times without token transfer
- **Challenges**: ensuring freedom from deadlock and starvation
• **Token will NOT be passed unless needed** (contingency)
• Boolean variable *haveToken* enables entering CS (see p2)
• *token* message type includes array *granted* containing ticket number of each node the *last time* it had permission to CS
• Each node stores array *requested* containing ticket numbers from *the last request messages* sent by the other nodes
  • Different nodes can have different values in *requested*
• *granted* enables determining what outstanding *request* messages have not yet been satisfied
• Requests from Becky and Danielle sent after the last time they were granted permission to enter CS
• If Chloe holds token and is not in CS (or after leaving CS), must send it to one of them - this prevents starvation
## Algorithm 10.3: Ricart-Agrawala token-passing algorithm

| 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
## Main

```plaintext
loop forever

p1: non-critical section

p2: if not haveToken

p3: myNum ← myNum + 1

p4: for all other nodes N

p5: send(request, N, myID, myNum)

p6: receive(token, granted)

p7: haveToken ← true

p8: inCS ← true

p9: critical section

p10: granted[myID] ← myNum

p11: inCS ← false

p12: sendToken
```
Algorithm 10.3: Ricart-Agrawala token-passing algorithm (continued)

**Receive**

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p13</td>
<td>receive(request, source, reqNum)</td>
</tr>
<tr>
<td>p14</td>
<td>requested[source] ← max(requested[source], reqNum)</td>
</tr>
<tr>
<td>p15</td>
<td>if haveToken and not inCS</td>
</tr>
<tr>
<td>p16</td>
<td>sendToken</td>
</tr>
</tbody>
</table>
Current ticket number of a node incremented in $p3$ and sent in request messages to enable updating array requested in all other nodes.

When a node completes CS its array granted is updated and sent in token message.

Limitation: the queue of waiting processes is transmitted in token message - inefficient when many nodes (but still more efficient than original Ricart-Agrawala algorithm, due to the lower overhead of sending 1 token message).
**RICART-AGRAWALA TOKEN-PASSING ALGORITHM - MUTUAL EXCLUSION**

- Theorem: *Mutual exclusion satisfied*
- Proof: Process only enters CS if `haveToken` is *true*

Initially `haveToken` is *true* in only 1 node and can be changed only if `token` message is received

`token` message is sent only by the node where `haveToken` is *true*, but immediately after this node sets `haveToken` to *false*

Thus, it is impossible for 2 nodes to have `haveToken` as *true* at the same time

∎
• Theorem: Free from deadlock

• Proof: Node that wants to enter CS but cannot, must be blocked waiting at `receive(token, granted)`

For each such node \( i \), eventually \( \text{requested}[i] > \text{granted}[i] \) in the node with token

If node with token not in CS: `token` message will be sent (see \( p16 \)) to 1 of the blocked nodes when its request is received

If node with token in CS: assuming progress of CSes, token will eventually be sent in \( p12 \)
RICART-AGRAWALA TOKEN-PASSING ALGORITHM - STARVATION

- Possible starvation (!) due to arbitrary selection of “some such $N$” in `sendToken`
- Version without starvation: Maintain ID of the last process granted and start searching from there - `sendToken` becomes:

<table>
<thead>
<tr>
<th>Algorithm Ans.10.1: Ricart-Agrawala token passing (no starvation 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer start ← 0</td>
</tr>
<tr>
<td>p1: for $l$ from 1 to NODES</td>
</tr>
<tr>
<td>p2: start = start + $l$ mod NODES</td>
</tr>
<tr>
<td>p3: if request[start] &gt; granted[start]</td>
</tr>
<tr>
<td>p4: send(token, start, granted)</td>
</tr>
<tr>
<td>p5: haveToken ← false</td>
</tr>
<tr>
<td>p6: break</td>
</tr>
</tbody>
</table>
Nielsen-Mizuno token-passing algorithm is based on passing a small token in a set of virtual spanning trees implicitly constructed by the algorithm.

More efficient than Ricart-Agrawala token passing.

Requires understanding of virtual spanning trees, which will be explained (from textbook Chapter 11) in a future lecture.

Optional task for you: After virtual data structures are taught in a future lecture, revisit Ben-Ari’s textbook Chapter 10 and independently study Nielsen-Mizuno token-passing algorithm.
NEXT TIME...
(PREVIEW HIGHLIGHTS)

From additional material NOT in the textbook!
MAIN TOPICS IN THE NEXT LECTURE…

(Not in the textbook!)

• Ricart-Agrawala algorithm demo in DAJ
• Revision of message-passing using CSP channels
• The actor model for message-passing concurrency
• Brief overview of some other distributed message-passing and distributed shared memory paradigms
• Notes on other concepts (e.g. future/promise, stream, ...) for programming asynchronous distributed systems