Advanced Network Security

3. Distributed Algorithms: Mutual Exclusion

Jaap-Henk Hoepman

Digital Security (DS)
Radboud University Nijmegen, the Netherlands
@xotoxot // ⚡ jhh@cs.ru.nl // 🌐 www.cs.ru.nl/~jhh
Victorian swimming pool
Mutual exclusion

- Suppose nodes occasionally need access to a shared resource, but accessing it simultaneously creates problems
  - E.g. printing a document, updating a record in shared memory

- We need a protocol that allows nodes to request access, and that guarantees that such nodes eventually get exclusive access.
  - Nodes call `enter()` to request access
  - Nodes call `exit()` to release the resource

```plaintext
while true
do enter()
   /* critical section */
extit()
   /* non critical section aka remainder section */
```
Mutual exclusion

- **A mutual exclusion protocol has the following properties**
  - **Mutual exclusion**: there is at most one node in the *critical section*
  - **Progress**: if there is at least one node enters, and the critical section is empty, then one of these nodes will eventually get access to the critical section
  - **No starvation**: if a node enters, and if all nodes that get access to the critical section release it, then it will eventually get access

- **Assumption:**
  - a fair scheduler, and
  - atomic shared variables
How could you solve mutual exclusion?

- **Enter**
  - atomic
  - while $i = 0$ do (wait)
  - $c := c - 1$

- **Share**
  - $i$
  - integer
  - # of allowed

- **Exit**
  - $i := i + 1$

---

The diagram illustrates a situation where two processes, labeled $a$ and $b$, attempt to enter a critical section (C.S.) simultaneously. The process labeled $a$ sets $i := 1$, while the process labeled $b$ sets $i := 0$. However, this does not work because both processes cannot enter the critical section at the same time.

- $i := i - 1$
- $i > 0$

Both processes are blocked until one allows the other to proceed, indicating the need for a mutual exclusion mechanism to prevent this race condition.
Mutual exclusion using message passing (1)

- Using logical clocks [Lamport]
  - Every message $m$ carries a timestamp $T_m$ (equal to the logical clock value $C_i(a)$ of the send event $a$)
  - Recall this induces a total order $\Rightarrow$
  - Every node maintains a request queue of $\langle timestamp, node \rangle$ pairs ordered by $\Rightarrow$ as well, initially $\langle -1,0 \rangle$ for each node
  - As we will see later, this means node 0 initially holds the resource
Mutual exclusion using message passing (2)

**Rules**

- (1) To request a resource, node $i$ sends a request message to all other nodes, including itself. These messages include a timestamp.
- (2) When a node $j$ receives a request message with timestamp $T_m$ from node $i$:
  - (i) Node $j$ adds $(T_m, i)$ to its request queue.
  - (ii) Node $j$ sends a (timestamped) acknowledgement message back to $i$.
- (3) To release a resource, node $i$ sends a release message to all other nodes, including itself.
- (4) When a node $j$ receives a release message with (timestamp $T_m$) from node $i$ it removes any $(*, i)$ messages from its request queue.
- (5) A node $i$ is granted access to the resource if
  - (i) Node $i$’s request queue contains $(T, i)$ ordered (using $\Rightarrow$) before any other elements $(T', j)$ in the request queue.
  - (ii) Node $i$ has received a message from all other nodes with a timestamp $T'' > T$.
The protocol in action

\[
\left\langle 10, i \right\rangle
\]
Why does this work? (1)

- Mutual exclusion
Why does this work? (1)

- **Mutual exclusion**
  - Rule 1 and 2 ensure that resource requests are added to all queue’s
  - 5.ii guarantees that a node must have learned about earlier requests before honouring its own
  - Requests are only removed from the queue when the corresponding node releases it (and sends release messages to all nodes) due to rule 3 and 4
Why does this work? (2)

- Progress

- No starvation

If we use leaders

One Request Queue

\( \text{leader} \)

\( \text{crashers} \)

\( \Rightarrow \text{system freezes} \)

- If a node crashes while entering, exiting or in the C.S.
- If a node crashes in the remainder section

\( \Rightarrow \text{system stays alive} \)

 Logical clocks
Why does this work? (2)

- **Progress:** in fact, requests are honoured in the order in which they are made
  - Follows from the fact that $\Rightarrow$ extends $\rightarrow$, requests $(T, i)$ are ordered by $\Rightarrow$ and served in that order because of rule 5.i and the above

- **No starvation:** every request is eventually honoured
  - For every request, a node receives an acknowledgement (rule 2), hence rule 5.ii is eventually satisfied
  - If every node eventually releases the resource, then rule 3 and 4 guarantee that eventually any older timestamps are removed from the request queue
Mutual exclusion with shared memory

- An easy solution...

\[
\text{testandset}(x) \quad \text{return } x
\]
\[
\text{if } x=0 \text{ then } x=1 \quad \text{One atomic operation}
\]

\[
\text{while } \neg \text{testandset}(x) \text{ do wait}
\]
\[
(\# \text{critical section } x)
\]
Mutual exclusion with normal shared memory: first try

- Two shared variables
  - $flag_0$ written by 0 and read by 1
  - $flag_1$ written by 1 and read by 0

- Protocol
  
  \[
  flag_i = \text{false} \\
  \text{while true} \\
  \quad \text{while } flag_{1-i} \text{ do } /* \text{wait} */ \\
  \quad \quad flag_i = \text{true} \\
  \quad /* \text{critical section} */ \\
  \quad flag_i = \text{false}
  \]

- What can go wrong?

- What if we first set the flags, before testing their value?
Mutual exclusion: Lamport’s bakery algorithm

- Each node $i$ maintains two shared variables that it writes and that all other nodes can read
  - $C[i].num$, unbounded
  - $C[i].choosing$

- **Idea: take numbered ticket like in the bakery**
  - Except that you have to ask everyone in the shop what their number is, and take the maximum $+ 1$
  - And you should wait for people that haven’t picked a number yet

- Lowest number is next allowed in the critical section
Mutual exclusion: Lamport’s bakery algorithm

\[ \text{num}[i] = 0 \]
\[ \text{choosing}[i] = \text{false} \]

\textbf{while true}
\begin{align*}
&\text{do}
&\text{choosing}[i] = \text{true}
&\text{num}[i] = 1 + \max_j \text{num}[j]
&\text{choosing}[i] = \text{false}
&\textbf{for} \ j \neq i
&\text{do while choosing}[j] \textbf{ do} /* wait */
&\text{while} \ (\text{num}[j] > 0) \land ((\text{num}[j], j) < (\text{num}[i], i)) \textbf{ do} /* wait */
&\text{/* critical section */}
&\text{num}[i] = 0
\end{align*}
**Proof of bakery algorithm: mutual exclusion**

- **Lemma 1:** If $i$ in C.S. and there is a $k$ s.t. $\text{num}[k] \neq 0$ then
  $$(\text{num}[k], k) > (\text{num}[i], i)$$
  
  - $i$ checked status of $k$ in for/while loop, at which point either
    - $\text{num}[k] = 0$ (or $(\text{num}[k], k) > (\text{num}[i], i)$ already)
    - So somewhere $\text{num}[k]$ became $\neq 0$

```
node k

write num[k] when ticket value computed when entering

read choosing[k] = false also in the test to enter C.S.

write num[k]
```

```
node i

write num[i]

read num[k] (when i sees num[k] = 0)
in the test to enter C.S.

read num[k]

read value written by i when entering

written ticket value must be bigger than this.

i in C.S. with num[k] #0
```
Proof of bakery algorithm: mutual exclusion

Lemma 1: if \(i\) in C.S. and there is a \(k\) s.t. \(\text{num}[k] \neq 0\) then \((\text{num}[k], k) > (\text{num}[i], i)\)

- Before entering, node \(i\) first waited until \(\text{choosing}[k] = \text{false}\) and then waited until either \((\text{num}[k] = 0)\) or \(( (\text{num}[k], k) > (\text{num}[i], i))\)

- If at that time \(\text{num}[k] = 0\), then by assumption that now \(\text{num}[k] \neq 0\), node \(k\) must have changed \(\text{num}[k]\) after node \(i\) read it (i.e. \(\text{read num}[k] \rightarrow \text{write num}[k]\)). Node \(i\) set its current value of \(\text{num}[i]\) before reading \(\text{choosing}[k] = \text{false}\) and reading \(\text{num}[k]\) (i.e. \(\text{write num}[i] \rightarrow \text{read choosing}[k] = \text{false} \rightarrow \text{read num}[k]\)). But node \(k\) sets \(\text{choosing}[k] = \text{true}\) before reading \(\text{num}[i]\) and writing \(\text{num}[k]\) (i.e. \(\text{write choosing}[k] = \text{true} \rightarrow \text{read num}[i] \rightarrow \text{write num}[k]\)).

- This can only happen if \(\text{read choosing}[k] = \text{false} \rightarrow \text{write choosing}[k] = \text{true}\). Hence \(\text{write num}[i] \rightarrow \text{read num}[i]\) so node \(k\) must have seen this value for \(\text{num}[i]\) when computing a ticket. By the protocol \(k\) sets \(\text{num}[k]\) to a larger value

- If instead \((\text{num}[k], k) > (\text{num}[i], i)\) then either node \(k\) did not change the value, or it entered again and by the same argument as above sets \(\text{num}[k]\) to a larger value
Proof of bakery algorithm: mutual exclusion

- **Lemma 1:** if \( i \) in C.S. and there is a \( k \) s.t. \( \text{num}[k] \neq 0 \) then \((\text{num}[k], k) > (\text{num}[i], i)\)

- **Lemma 2:** for all \( i \), \( \text{num}[i] \geq 0 \)
  - Follows from the protocol

- **Lemma 3:** if \( i \) in C.S. then \( \text{num}[i] > 0 \)
  - Follows from lemma 2 and the fact that \( i \) chooses \( \text{num}[i] = 1 + \max_j \text{num}[j] \)

- **Theorem:** the bakery protocol satisfies mutual exclusion
  - Suppose not. Then for \( i \neq j \) we have \( \text{num}[i] > 0 \) and \( \text{num}[j] > 0 \) by lemma 3 and then both \((\text{num}[i], i) > (\text{num}[j], j)\) and \((\text{num}[j], j) > (\text{num}[i], i)\) by lemma 1. A contradiction.
Proof of bakery algorithm: progress

- Suppose C.S. is empty and there are nodes entering
  - Note: the exit() protocol will eventually be completed
- Every node that enters gets a ticket
- Let \( i \) be the node with minimal \((num[i], i)\)
- It will never see nodes with smaller tickets
- It only needs to wait for nodes that choose, but these always complete after some number of steps (non blocking operation)
- It eventually enters the C.S., so we have progress.
Proof of bakery algorithm: no starvation

- In general we have that if progress holds, and if any node $i$ is waiting at most a finite number of other nodes can enter the C.S, then no starvation holds.

- Lemma 4: Let $i$ have a ticket. Any node entering after that will have a bigger ticket. Hence at most $n - 1$ nodes can have a smaller ticket and beat node $i$.

- Because progress holds and lemma 4 holds, no starvation then also holds.