Fair Synchronization in the Presence of Process Crashes and its Weakest Failure Detector

C. Delporte†, H. Fauconnier†, M. Raynal⋆

†LIAFA, Université Paris 7 Diderot, Paris, France

⋆Institut Universitaire de France

& IRISA, Université de Rennes, France

& Polytechnic University, Hong Kong

Summary

- Computing Model: Processes and Concurrent Objects
- Progress properties
- Failure-free fair synchronization: definition + algorithm
- The failure detector approach and QP (quasi-perfect)
- QP-based crash-prone fair synchronization
- Properties of QP
- Conclusion: target → multicores

Part I

MODEL and DEFINITIONS

Asynchronous process model

- A set Π of n processes $p_1, \ldots, p_n$
- Timing model: Asynchrony: No upper bound on the time required to execute a computation step
- Failure-free model (for the moment)
- Cooperation model: Read/Write registers only

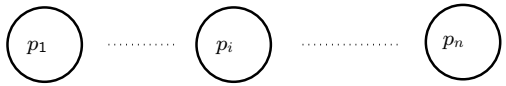
Part II

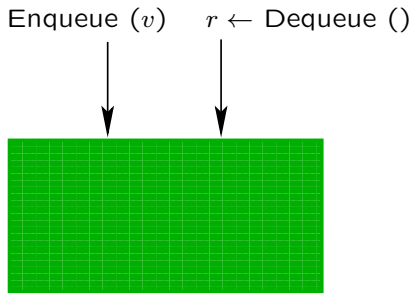
Concurrent object

Progress conditions

Concurrent Object

An object (defined by a sequential specification) that can be accessed by different processes





Concurrent object: Progress conditions

Usual progress conditions (liveness) properties

- Deadlock-freedom: OBJECT POINT OF VIEW

  when one or more processes invoke concurrently operations on a given object, at least one invocation terminates (the object is used)

- Starvation-freedom: PROCESS POINT OF VIEW
  any process returns from any invocation of an object operation (any process is served)

Classical implementation of progress conditions

- based on locks (mutex)
  - ⋆ locks are used to ensure both
    - ∗ object internal representation consistency
    - ∗ progress conditions
      - · it is possible to build deadlock-free locks from atomic registers
      - · it is possible to build starvation-free locks from deadlock-free locks and atomic registers
- can create waiting chains

Fairness

- Let p be process and O a concurrent object
- A fairness property restricts the concurrency pattern in which processes terminate their operations on O while p has not yet terminated its own invocation
- Fairness has mainly been adressed in the context of resource allocation algorithms, with specific solutions for each problem
- Starvation-freeedom $\not\Rightarrow$ fairness

  Starvation-freedom does not prevent a process from being *delayed during an arbitrarily long period of time* during which other processes execute an arbitrarily large (but finite) number of operations on the same object

Fairness $\Rightarrow$ Starvation-freedom $\Rightarrow$ Deadlock-freedom

Mutex-free progress conditions

- Non-blocking and wait-freedom are the counterparts of deadlock-freedom and starvation-freedom, respectively, for implementations of concurrent objects in which mutual exclusion (locks) are forbidden
- In these implementations no part of the internal representation of the object is protected by a lock

  Hence, processes that have invoked object operations can simultaneously access the internal representation of the object

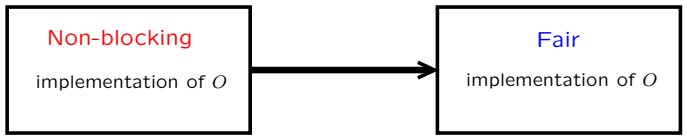
Wait-freedom $\Rightarrow$ Non-blocking $\Rightarrow$ Obstruction-freedom
Wait-freedom $\Rightarrow$ Starvation-freedom

Part III

Taubenfeld's
fair synchronization object
(DISC 2013)

Generic construction

- Let O be a concurrent object
- Assume we have a non-blocking implementation
- Taubenfeld's transformation



From non-blocking object operations to fair object opera

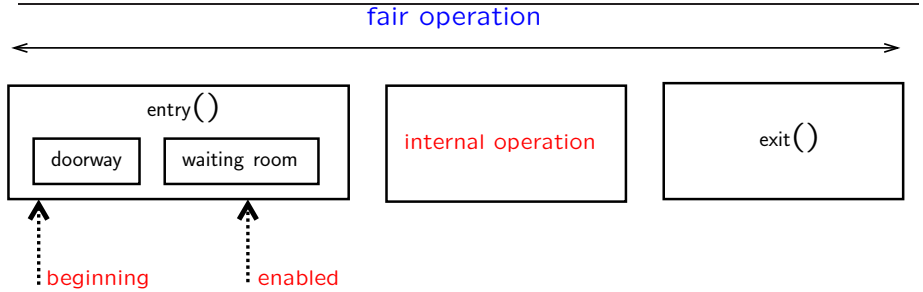
Object operations $op_1()$, $op_2()$, etc. (internal operations)

operation fair_$op_x()$ **is**
entry(); $op_x()$; exit()
end operation.

fair_$op_x()$: fair operation obtained from the internal object operation $op_x()$

entry() and exit(): additional control operations, shared by all fair operations on the object

Structure



- The control operation entry() is made up of two parts: a doorway and a waiting room
  - ⋆ The code of the doorway and exit() is wait-free
  - ⋆ The waiting room is the only part of code in which the invoking process is allowed to wait

Fair synchronization problem (1)

- Deadlock-freedom progress: If no process is executing an internal operation, and one or several processes invoke fair operations, then at least one of them will execute the corresponding internal operation

Fair synchronization problem (2)

- Fairness:
  - ⋆ A beginning process cannot execute twice an internal operation op(), before a waiting process completes executing its internal operation and exit code
  - ⋆ First-in/first-enabled: No beginning process can become enabled before an already waiting process becomes enabled
- Concurrency: All the waiting processes which are not enabled become simultaneously enabled

Generic Fair Synchro: Internal control objects

- $NEXT\_BATCH \in \{0,1\}$: next batch number

  $NEXT\_BATCH = x$: priority is currently given to the batch of invocations whose batch number is $1-x$

  $STATE[1..n]$: array of shared registers, where $STATE[i]$ can be written only by p_i

  $STATE[i]$: current state of p_i with respect to fair synchronization, $STATE[i] \in \{0,1,2,3\}$

  - $STATE[i] = 3$: p_i is not interested
  - $STATE[i] = 2$: p_i has invoked a fair object operation but has not yet been assigned a batch $b \in \{0,1\}$
  - $STATE[i] = b$ where $b \in \{0,1\}$: p_i has invoked a fair object operation and has been assigned the batch b

Abstract waiting predicate

$$(STATE[i] \neq NEXT\_BATCH) \vee \big((\forall\ j:\ (STATE[j] = 3)\ \vee\ (STATE[j] = STATE[i])\big)$$

- $(STATE[i] \neq NEXT\_BATCH)$ states that p_i has priority because its batch number (recorded in $STATE[i]$) is equal to the current priority batch (the number of which is $1 - NEXT\_BATCH$)
- The second predicate states that for each p_j, p_j is either not interested in accessing the object, or in the same batch as p_i

The fair synchronization algorithm

operation entry() **is**
 $STATE[i] \leftarrow 2$;
 $aux \leftarrow NEXT\_BATCH$; $STATE[i] \leftarrow aux$;
 for j **from** 1 **to** n **do**
  if $(STATE[i] \neq NEXT\_BATCH)$ **then** exit loop **end if**;
  wait $(STATE[j] \neq 2)$;
  if $(STATE[j] = 1 - STATE[i])$
   then wait $\big((STATE[j] \neq 1 - STATE[i]) \vee (STATE[i] \neq NEXT\_BATCH)\big)$
  end if;
 end for.

operation exit() **is**
 $NEXT\_BATCH \leftarrow (1 - STATE[i])$;
 $STATE[i] \leftarrow 3$.

PART IV

When there are process crashes

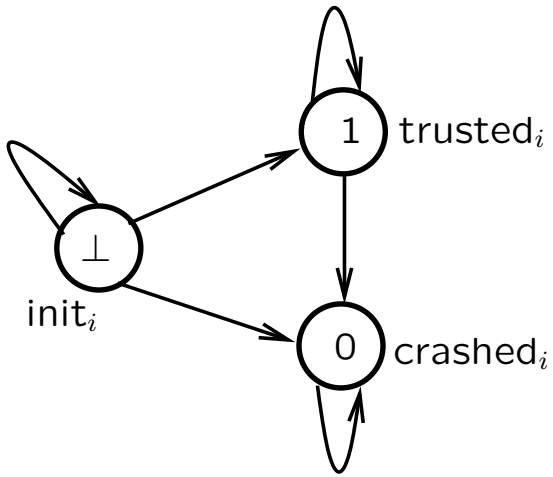
Crash failures and failure detector

- Computation model
  - ⋆ Asynchrony plus
  - ⋆ Any number of processes may crash (unexpected halt)
- The failure detector approach
  - ⋆ Provide each process with information on failures that allow them to solve the problem
  - ⋆ Find the weakest assumption on failures that allow the problem to be solved
  - ⋆ Theory vs practice

The quasi-perfect (QP) failure detector (1)

- Weakened version of $\mathcal{P}$ (the perfect failure detector)
- $\mathcal{P} \succ \mathcal{QP} \succ \Diamond\mathcal{P}$
- Each process p_i is has two read-only sets (initially empty)
  - ⋆ trusted_i and
  - ⋆ crashed_i
- $\text{init}_i = \Pi \setminus (\text{trusted}_i \cup \text{crashed}_i)$
- $\mathcal{C}$ = set of correct processes
- $\mathcal{F}$ = set of faulty processes
- $xxx_i(\tau)$ = value of xxx_i at time τ

QP automaton at a process p_i



QP: definition (1)

- $\forall\ \tau$: $\text{trusted}_i(\tau) \cap \text{crashed}_i(\tau) = \emptyset$.
- $p_j \in \text{trusted}_i(\tau) \Rightarrow$
  $\big(\forall\ \tau' \geq \tau$: $p_j \in \text{trusted}_i(\tau') \cup \text{crashed}_i(\tau')\big) \wedge$
  $\big(\forall p_k \in \mathcal{C}\ \exists \tau'$: $p_j \in \text{trusted}_k(\tau') \cup \text{crashed}_k(\tau')\big)$.

  A trusted process has to be eventually observed by all correct processes
- $p_j \in \text{crashed}_i(\tau) \Rightarrow j \in \mathcal{F}(\tau)$.
  crashed_i contains only crashed processes
- $p_j \in \text{crashed}_i(\tau) \Rightarrow \big(\forall\ \tau' \geq \tau$: $j \in \text{crashed}_i(\tau')\big)$.
  "Recognized as crashed" is a stable property

QP: definition (2)

- $p_j \in \mathcal{F}(\tau) \Rightarrow \big(\exists \tau' \geq \tau\text{: } j \notin \text{trusted}_i(\tau')\big)$.
  Eventually, any faulty process $\notin$ trusted$_i$
- $p_j \in \mathcal{C} \Rightarrow \big(\exists \tau\text{: } j \in \text{trusted}_i(\tau)\big)$.
  Eventually, any correct process $\in$ trusted$_i$

QP-based fair synchronization (1)

init: $NEXT\_BATCH \in \{0,1\}$ initialized arbitrarily;
$STATE[i] \in \{0,1,2,3\}$ initialized to 3;
% $MONITOR[i,j]$ monitoring of p_i by p_j
for each $1 \leq i,j \leq n$ **do** $MONITOR[i,j] \leftarrow \perp$ **end for**;
function crashed(j) **is** $(\exists\ k : MONITOR[j,k] = 0)$ **end function**.

background task mn_i **is**
  repeat forever
    for j **from** 1 **to** n **do**
      if $(p_j \in \text{trusted}_i)$ **then** $MONITOR[j,i] \leftarrow 1$ **end if**;
      if $(p_j \in \text{crashed}_i)$ **then** $MONITOR[j,i] \leftarrow 0$ **end if**
  end repeat.

QP-based fair synchronization (2)

operation entry() **is**
 wait $(\exists\ x : MONITOR[i,x] = 1)$;
 $STATE[i] \leftarrow 2$;
 $aux \leftarrow NEXT\_BATCH$; $STATE[i] \leftarrow aux$;
 for j **from** 1 **to** n **do**
   if $(STATE[i] \neq NEXT\_BATCH)$ **then** exit the loop **end if**;
   wait $\big((STATE[j] \neq 2) \vee \text{crashed}(j)\big)$;
   if $(STATE[j] = 1 - STATE[i])$
     then wait crashed$(j) \vee$
     $STATE[j] \neq 1 - STATE[i] \vee STATE[i] \neq NEXT\_BATCH$
   end if;
 end for.

operation exit() **is** as before.

QP's fundamental properties

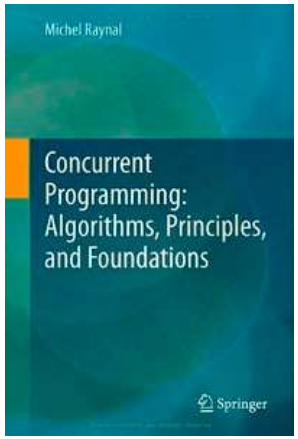
- There is an algorithm (described in the proceedings) that extracts a failure detector satisfying the properties of $\mathcal{QP}$ from any algorithm solving the fair synchronization problem in the asynchronous read/write systems where any number of processes may crash
- $\mathcal{QP}$ is strictly stronger than the weakest failure detector that allows mutual exclusion to be solved in asynchronous read/write systems where any number of processes may crash
- $\mathcal{QP}$ allows mutual exclusion to be solved in asynchronous message-passing systems where a majority of processes are correct

CONCLUSION

What do we have visited?

- The notion of a fair synchronization object
- Systematic and generic constructions of fair object from non-blocking implementations of the same object
- Weakest failure detector for implementing a fair synchronization object

More in



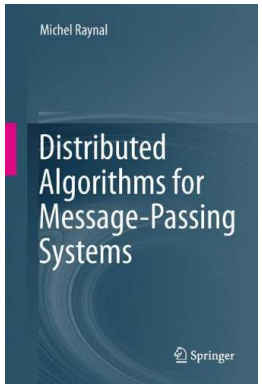
Concurrent Programming: Algorithms, Principles and Foundations

by Michel Raynal

Springer, 531 pages, 2013

ISBN: 978-3-642-32026-2

Yet another book on Message-passing Algorithms



Distributed Algorithms for Message-Passing Systems

by Michel Raynal

Springer, 515 pages, 2013

ISBN: 978-3-642-38122-5-2