SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

CONCURRENT SYSTEMS
LECTURE 8

Prof. Daniele Gorla

Enhancing Liveness Properties

For MUTEX-based concurrency we saw that a weak liveness property (deadlock freedom) can be always enhanced to a stronger one (bounded bypass)

We want to do the same in the framework of MUTEX-free concurrency

Contention manager: is an object that allows progress of processes by providing contention-free periods for completing their invocations. It provides 2 operations:

- need_help(i) : invoked by pi when it discovers that there is contention
- stop_help(i) : invoked by pi when it terminates its current invocation

Enriched implementation: when a process realizes that there is contention, it invokes need_help; when it completes its current operation, it invokes stop_help.

REMARK: this is different from lock/unlock because in this framework we allow (fail-stop) failures, that can also happen during the contention-free period

→ the contention-free period always terminates

PROBLEM: to distinguish a failure from a long delay, we need objects called ***failure detectors***, that provide processes information on the failed processes of the system.

→ according to the type/quality of the info, several F.D.s can be defined

From obstruction-freedom to non-blocking

Eventually restricted leadership: given a non-empty set of process IDs X, the failure detector Ω_X provides each process a local variable ev_leader(X) such that

1. (*Validity*) ev_leader(X) always contains a process ID
2. (*Eventual leadership*) Eventually, all ev_leader(X) of all non-crashed processes of X for ever contain the same process ID, that is one of them

REMARK: the moment in which all variables contain the same leader is unknown

```
NEED_HELP[1..n] : SWMR atomic R/W boolean registers init at false
```

```
need_help(i) :=
   NEED_HELP[i] ← true
   repeat
        X ← {j : NEED_HELP[j]}
   until ev_leader(X) = i
```

```
stop_help(i) :=
   NEED_HELP[i] ← false
```

Thm.: the contention manager just seen transforms an obstr.-free implementation into a non-blocking enriched implementation.

Proof:
By contr., $\exists\ \tau$ s.t. $\exists$ many (> 0) op.'s invoked concurrently that never terminate
Let Q be the set of proc.'s that performed these invocations.
- By enrichment, eventually NEED_HELP[i]=T ($\forall$ i $\in$ Q) forever
- Since crashes are fail-stop, eventually NEED_HELP[j] is no longer modified ($\forall$ j $\notin$ Q)
  → $\exists\ \tau' \geq \tau$ when all proc.'s in Q compute the same X

OBS.: Q $\subseteq$ X (it is possible that pj sets NEED_HELP[j] and then fails)

By def. of Ω_X , $\exists\ \tau'' \geq \tau'$ s.t. all proc.'s in Q have the same ev_leader(X)
→ the leader belongs to Q, since it cannot be failed
→ this is the only process allowed to proceed
→ because run in isolation, it eventually terminates (bec. of obstr-freedom)

On implementing Ω

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

It can be proved that there exists no wait-free implementation of Ω in an asynchronous system with atomic R/W registers and any number of crashes

→ crashes are indistinguishable from long delays

→ need of timing constraints

1. $\exists$ time τ_1, time interval Δ and correct process p_L s.t. after τ_1 every two consecutive writes to a specific SWMR atomic R/W register by p_L are at most Δ time units apart one from the other

2. Let t be an upper bound on the number of possible failing processes and f the real number of processes failed (hence, $0 \leq f \leq t \leq n-1$, with f unknown and t known in advance).

   Then, there are at least $t-f$ correct processes different from p_L with a timer s.t.

   $\exists$ time τ_2 $\forall$ time interval δ , if their timer is set to δ after τ_2 it expires at least after δ

REMARK: τ_1, τ_2, Δ and p_L are all unknwon

On implementing Ω

IDEA:

- PROGRESS[1..n] is an array of SWMR atomic registers used by proc's to signal that they're alive
  - → pi regularly increases PROGRESS[i]
  - → p_L eventually increases PROGRESS[L] every Δ time units at the latest
- pi suspects pj if pi doesn't see any progress of pj after a proper time interval (to be guessed) set in its timer
- The leader is the least suspected process, or the one with smallest/biggest ID among the least suspected ones (if there are more than one)
  - → this changes in time, but not forever

Guessing the time duration for suspecting a process:

- SUSPECT[i,j] = #times pi has suspected pj
- For all k, take the t+1 minimum values in SUSPECT[1..n , k]
- Sum them, to obtain S_k
- The interval to use in the timers is the minimum S_k
  - → it can be proved that this eventually becomes $\geq \Delta$

From obstruction-freedom to wait-freedom

Eventually perfect: the failure detector $\Diamond P$ provides each process pi a local variable $suspected_i$ such that

1. (*Eventual completeness*) Eventually, $suspected_i$ contains all the indexes of crashed processes, for all correct pi
2. (*Eventual accuracy*) Eventually, $suspected_i$ contains only indexes of crashed processes, for all correct pi

Def.: FD1 is **stronger** than FD2 if there exists an algorithm that builds FD2 from instances of FD1 and atomic R/W registers

Prop.: $\Diamond P$ is stronger than Ω_X .

Proof:

Forall i

- $i \notin X \rightarrow ev\_leader_i(X)$ is any ID (and may change in time)
- $i \in X \rightarrow ev\_leader_i(X) = \min((\Pi \setminus suspected_i) \cap X)$

where Π denotes the set of all proc. IDs

From obstruction-freedom to wait-freedom

Ω_X is NOT stronger than $\Diamond$P (so, $\Diamond$P is strictly stronger).

One possible idea (WRONG!) is

- Run Ω_Π that eventually fixes $p_{\ell 1}$
- After this, run $\Omega_{\Pi \setminus \{\ell 1\}}$ that eventually fixes $p_{\ell 2}$
- After this, run $\Omega_{\Pi \setminus \{\ell 1, \ell 2\}}$ that eventually fixes $p_{\ell 3}$
- …

This eventually calculates the set of all non-crashed proc.'s

→ PROBL.: we cannot know when a leader is elected (permanently)

The formal proof consists in showing that, if Ω was stronger than $\Diamond$P, then consensus would be possible in an asynchronous system with crashes and atomic R/W registers.

From obstruction-freedom to wait-freedom

We assume a *weak timestamp generator*, i.e. a function such that, if it returns a positive value t to some process, only a finite number of invocations can obtain a timestamp smaller than or equal to t

```
TS[1..n] : SWMR atomic R/W registers init at 0

need_help(i) :=
   TS[i] ← weak_ts()
   repeat
        competing ← {j : TS[j]≠0 ∧ j ∉ suspected_i}
        ⟨t,j⟩ ← min{⟨TS[x],x⟩ | x ∈ competing}
   until j = i


stop_help(i) :=
   TS[i] ← 0
```

<u>Thm.:</u> the contention manager just seen transforms an obstr-free implementation into a wait-free enriched implementation.

Proof:

By contr., ∃ an invocation of a correct pi that never terminates; let ti be its timestamp

→ choose the minimum of such ⟨ti,i⟩

By constr. of weak_ts(), the set of invocations smaller than ⟨ti,i⟩ (call it I) is finite

- For every invocation ∈ I from a process pj that crashes during its execution
  - → pi will eventually and forever suspect pj (i.e., j ∈ $suspected_i$)
  - → eventually, j ∉ $competing_i$ and, thus, won't prevent pi from proceeding
- Since ⟨ti,i⟩ is the minimum index of a non-terminating invocation
  - → all invocations ∈ I of correct processes terminate
  - → if such processes invoke need_help() again, they obtain greater indexes
  - → eventually I gets emptied

Since pi is correct, eventually (for all pk correct):

- i ∉ $suspected_k$
- ⟨ti,i⟩ = min{⟨TS[x],x⟩ | x \in $competing_k$}

Hence, the invocation with index ⟨ti,i⟩ will eventually have exclusive execution

→ because of obstr.-freedom it eventually terminates

OBS: since non-blocking implies obstr.-fr., the Thm holds also for non-blocking impl.

On implementing $\Diamond P$:

- Every non-failed process has eventually an upper bound on the write delay
- By properly setting timers, eventually crashed processes are distinguished from the non-crashed ones by looking at the suspicions: for the crashed ones, this numbers increases indefinitely; for non-crashed ones, some reset eventually happens.