Renaming
Renaming, Message Passing and Failure detectors
(dernier cours)
renaming

• \( p_0, p_1, \ldots, p_n : i \) is the index of \( p_i \)

• each process has an initial name : \textit{old\_name} from some set \([1..N]\) (with \( N \) big)
  • give a \textit{new\_name} in some set \([1..M]\) with \( M < N \)

\text{new\_name()} \text{ such that:}

• termination
• validity: each id returned by \text{new\_name()} is in set \([1..M]\)
• agreement: no two processes obtain the same new name
• index independence: the new name is independent of the index
• one shot : each process invoke \text{new\_name} at most once
• (long-lived: a process can repeatedly invoke a new name and the release the name)

(note that renaming is not colorless)
renaming with splitters...

(here renaming only with MWMR registers)

«splitters» [Moir, Anderson’95]:

validity: output Right, Down or Stop

solo: if a single process invokes the splitter only
Stop is possible

concurrency:

when k processes invoke the splitter:

at most k-1 processes obtain Right
at most k-1 processes obtain Down
at most one process obtains Stop

termination

(when a process obtains Stop -> gets a new name
associated to the splitter)
splitter

Code for the splitter with identity my_name:

```c
int closed=false

Last:=my_name
if(Closed) then return Right
else Closed:=True;
   if(Last=my_name)
      then return(Stop)
    else return(Down)
```

### Diagram

![Diagram of splitter](image-url)
Renaming

- Grid of $n(n+1)/2$ splitters (with an enumeration of the splitters on $\{1, \ldots, n(n+1)/2\}$)
- When a process obtains Stop, it chooses as new name the name of the splitter
- No process stops on the same splitter
- Every process stops on some splitter
Complexity

- \( \Theta(n^2) \) splitters
- then size is the set of new names: \( \Theta(n^2) \)
  - \( ((n+1)/2) \)-renaming
- two registers for each splitter (only MWMR registers): \( \Theta(n^2) \) registers
- (improvement [Aspnes’10] \( \Theta(n^{3/2}) \) splitters)
s:=1
forever do
    a.update(i,s)
    view:=a.snapshot()
    if view[j]=s for some j ≠ i then
        r= |{j with view[j] ≠ undef and j <=i}|
        s= r-th positive integer not in
            {view[j]| view[j] ≠ undef } 
    else return s
Proof

• r is the rank
• Because the rank is at most n and there are at most n-1 names used by the other processes, this always gives proposed names in the range [1..2n-1].
• **For uniqueness,** consider two process with original names i and j. Suppose that i and j both decide on s.
  • Then i sees a view in which a[i] = s and a[j] ≠ s, after it no longer updates a[i].
  • Similarly, j sees a view in which a[j] = s and a[i] ≠ s, after which it no longer updates a[j].
  • If i's view is obtained first, then j can't see a[i] ≠ s, but the same holds if j's view is obtained first. So in either case we get a contradiction, proving uniqueness.
Termination. Assume some set $S$ of processes run forever without picking a name, then $p$ in $S$ with minimal identity eventually picks a name.

- More precisely, say a process is \textit{trying} if it runs for infinitely many steps without choosing a name.
- Then in any execution with at least one trying process, eventually we reach a configuration where all processes have either finished or are trying.
- In some subsequent configuration, all the processes have written to the $a$ array at least once; after this point, any process reading the $a$ array will see the same set of original names and compute the same rank $r$ for itself.
  - Consider the trying process $i$ with the smallest original name, and suppose it has rank $r$. (all the other trying processes have a rank greater than $r$)
  - Let $F = \{ z_1 < z_2 \ldots \}$ be the set of "free names" that are not proposed in $a$ by any of the finished processes.
  - No trying process $j \neq i$ ever proposes a name in $\{ z_1 \ldots z_r \}$, This leaves $z_r$ open for $i$ to claim, provided the other names in $\{ z_1 \ldots z_r \}$ eventually become free.
    - $j$ will the only process to propose $z_r$
Better renaming?

Can we do better than 2n-1 (wait-free) renaming?

(Difficulty) result::

- no better than 2n-2 renaming if \( \left\{ \binom{n+1}{i+1} \mid 0 \leq i \leq \left\lfloor \frac{n-1}{2} \right\rfloor \right\} \)
- no better 2 than n-1 renaming in other cases
and message passing?
and message passing?

• message passing:
  
  • send/receive messages with asynchronous communication (each message sent by a process is eventually received by a correct process)
  
  • Process may crash (stop the execution)
  
  • process p is correct if it doesn’t crash (makes an infinity of steps)
  
  • t- resiliency at most t processes may crash
  
  • (complete communication graph, point to point communication)
Reliable Broadcast

• primitives \texttt{Rbcast Rdeliver}
  
  • \textit{Agreement:} if p correct \texttt{Rdeliver m} then every correct process \texttt{Rdeliver m}
  
  • \textit{Validity:} If p correct \texttt{Rbcast m} then p \texttt{Rdeliver m}
  
  • \textit{Integrity:} If p \texttt{Rdeliver m} then there is a process q that has \texttt{Rbcast m}
Algorithm for process $p$:

To execute $\text{Rbcast}(m)$
   send $(m)$ to $p$

$\text{Rdeliver}(m)$ occurs when
   upon $\text{receive}(m)$ do
      if has not previously executed $\text{Rdeliver}(m)$
      then
         send $(m)$ to all
         $\text{Rdeliver}(m)$
ABcast

• primitives \texttt{ABcast ABdeliver}:
  • RBcast properties:
    • \textbf{Total order}: If \( p \) and \( q \) ABdeliver \( m \) and \( m' \) then if \( p \) ABdelivers \( m \) before \( m' \) then \( q \) ABdelivers \( m \) before \( m' \)

ABCcast is « universal »: (very informal) (active replication)

• state machine replication:
  • any sequential state machine \( A \)
  • \( t+1 \) processes simulate \( A \)
  • each request is made by atomic broadcast
  • then we get a \( t \)-resilient implementation of \( A \)
Atomic Broadcast and Consensus

Consensus in message passing: decision algorithm such that:

- **termination**: all correct processes decide

- **validity**: if \( p \) decides \( v \) then \( v \) is an initial value of some process

- **agreement**: if \( p \) and \( q \) decide, they decide the same value
Algorithm for process $p$:

Initialization:
- $R_{Delivered} := \emptyset$
- $A_{Delivered} := \emptyset$

To execute $\text{Abcast}(m)$
- $\text{Rbcast}(m)$

$\text{Adeliver}(_) \text{ occurs when}$
- $\text{upon } R_{deliver}(m) \text{ do}$
  - $R_{Delivered} := R_{Delivered} \cup \{m\}$

$\text{do forever}$
- $A_{Undelivered} := R_{Delivered} - A_{Delivered}$
- $\text{if } A_{Undelivered} \neq \emptyset \text{ then}$
  - $k := k + 1$
  - $\text{propose}(k, A_{Undelivered})$
  - $\text{wait for } \text{decide}(k, \text{msgSet})$
  - $\text{batch}(k) := \text{msgSet} - A_{Delivered}$
  - A-deliver all messages in $\text{batch}(k)$ in some deterministic order
  - $A_{Delivered} := A_{Delivered} \cup \text{batch}(k)$

From Consensus and Reliable broadcast to Atomic Broadcast
Atomic broadcast and consensus are equivalent in message passing with crash failure.

Consensus is « universal » in message passing with crash failure.
Message Passing versus Shared memory

• simulating shared memory: with a majority of correct processes atomic registers may be implemented in asynchronous message passing models.

• (Note that a SWMR register enable to simulate a communication channel)

• Message-Passing and shared memory models are « equivalent » -with a majority of correct processes
Simulating shared registers in message passing

- simulation of a single-writer single-reader register (it is sufficient to simulate MWMR atomic registers)
  - assume we have a majority of correct processes
For the writer

to write($v$)

\[ seq := seq + 1 \]

send \((W, v, seq)\) to all

wait until receiving \([n/2] + 1\) messages \((ACK, seq)\)

For the reader

to read()

send\((R)\) to all

wait until receiving \([n/2] + 1\) messages \((V, v, s)\) such that \(s > seq\)

return \(val\)

such that \((V, val, S)\) has been received

and \(S\) is the max of the sequence number of received \(V\) messages

For all processes

when \((W, v, s)\) is received

if \(s > seq\) then

\[ val := v; seq := s \]

send \((ACK, s)\)

when\((R)\) is received

send \((V, val, seq)\) to \(p_r\)
A majority of correct processes is needed

partition argument:

• if \( n \leq 2t \) then we can partition the set of processes in two set \( S_1 \) and \( S_2 \) such that \( |S_1| \geq t \) and \( |S_2| \geq t \).

• Run \( A_1 \): all processes in \( S_1 \) are correct and all processes in \( S_2 \) are initially dead, \( p_0 \) invokes a \texttt{write}(1), at some time \( t_0 \) the \texttt{write} terminates

• Run \( A_2 \): all processes in \( S_2 \) are correct and all processes in \( S_1 \) are initially dead, \( p_1 \) in \( S_2 \) invokes a \texttt{read()} at time \( t_0 + 1 \) the \texttt{read} terminates at time \( t_1 \)

• Run \( B \): « merge » of \( A_1 \) and \( A_2 \) but no process crash. \texttt{write}(1) terminates before the \texttt{read()} and the \texttt{read} return 0
Come back to the consensus
How to circumvent these impossibility results?

• change the model:
  • synchronous model
  • partially synchronous models
• direct approach:
  • Dwork, Lynch and Stockmeyer Consensus in the presence of partial synchrony J. ACM 88
  • Dolev, Dwork and Stockmeyer On the minimal synchronism needed for distributed consensus. J. ACM 87
Failure Detector Chandra& Toueg

• PODC 91 - J. ACM 96
• Distributed oracles that give hints on the failure pattern (e.g. set of suspected processes)
• A FD is basically defined by:
  – a completeness property: actual detection of failure
  – an accuracy property: restrict the mistake that a FD can make
Failures pattern

- failures and failure pattern:
  - at time $t$: $p$ is faulty (dead, crashed: $p$ stops making steps) or is alive ($p$ eventually makes steps)
  - failure pattern (schéma de pannes): $F(t)$ is the set of faulty processes at time $t$; ($F(t) \subseteq F(t+1)$)
    - $p$ is faulty in $F$: $\exists \, t \, p \in F(t)$
    - $p$ is correct in $F$: $p$ is not faulty : $\forall \, t \, p \notin F(t)$

\[ F(t) = \emptyset \]
\[ F(t) = \{p\} \]
\[ F(t) = \{p, s\} \]
\[ F(t) = \{p, r, s\} \]
another schedule for the same failure pattern △
Failure detector[CT96]

- *failure detector*: oracle that may be invoked locally by processes depending only on failure pattern (not on the schedule!). At each step a process may invoke its failure detector module and gets an answer.

- depending only on the failure pattern: failure detector $\mathcal{X}$ is defined for each failure pattern $F$ by « histories » $H$

  - $H$ history of $\mathcal{X}$ for the failure pattern $F$: $H(p,t)$ is the output of $\mathcal{X}$ for process $p$ at time $t$ (if $p$ invokes failure detector $\mathcal{X}$ at time $t$, $p$ gets the answer $H(p,t)$)
Failure detectors

Examples of failure detectors with lists of suspected processes:

- $\mathcal{P}$ perfect FD: completeness + strong accuracy
- $\Diamond \mathcal{P}$: completeness + eventual strong accuracy
- $\Diamond \mathcal{S}$: completeness + eventual weak accuracy
- $\Omega$: outputs at each process one id of process (the expected leader):
  eventually $\Omega(p,t)$ is forever the same correct process for all processes (leader election)

$\mathcal{P}$ enables the consensus (synchronous rounds)

$\Diamond \mathcal{P}$, $\Diamond \mathcal{S}$, $\Omega$ enable the consensus (in shared memory or with a majority of correct processes in message passing)
Comparing failure detectors

Remarks:

• environment: set of of failure pattern e.g. less than m faulty processes (here implicitly any number of faulty processes)

• shared memory versus message passing: depending on environments, solvability is not (exactly) the same for message passing and shared memory
Comparing failure detectors

- Problem P is solvable with FD $\mathcal{X}$ iff there is a distributed algorithm using FD $\mathcal{X}$ that solves P

- Implementing a FD $\mathcal{X}$ may be (with more formal definitions) considered as a problem
Comparing failure detectors

\( A \preceq B \) (\( A \) is weaker than \( B \)) : there is a (distributed) algorithm with failure detector \( B \) that implements \( A \).

- \( B \) provides at least as much information about failures than \( A \)
- (intuitively: less information about failures in \( A \) than in \( B \))

\( A \prec B \) (\( A \) is strictly weaker than \( B \)) if \( A \preceq B \) and not \( B \preceq A \)

if \( A \preceq B \) and \( B \preceq C \) then \( A \preceq C \) and (with more formal definitions)

\( A \preceq A \)

- \( \preceq \) is a pre-order.
Weakest failure detector

As \( \preceq \) is a pre-order, failure detectors can be compared and given some set \( S \) of failure detectors \( M \) is the weakest for \( S \) iff for all \( X \in S \) \( M \preceq X \)

(from any \( X \), \( X \) contains at least as much information about failures than \( M \))

---

given a « problem » \( P \), failure detector \( W \mathcal{K} \) is a weakest failure detector for \( P \) iff

1. \( W \mathcal{K} \) can be used to solve \( P \)

2. for all \( X \) that can be used to solve \( P \) : \( W \mathcal{K} \preceq X \)

\( W \mathcal{K} \) encapsulates the minimum information about failures to solve problem \( P \)
Comparing problems with failure detectors

Given a « problem » $P$ (solvable by a FD) there is a weakest failure detector to solve $P$ [JT2008]

from the hierarchy of FD to hierarchy of problems:

- Given problems $P$ and $Q$: $P \preceq_{FD} Q$ iff $Q$ solvable with some FD $\mathcal{X}$ then $P$ is also solvable with $\mathcal{X}$

- $P \preceq_{FD} Q$ iff the $\text{WFD}(P) \preceq \text{WFD}(Q)$
Example: ◇ $S$ and $\Omega$

- $\Omega$: outputs at each process one id of process (the expected leader):
  eventually $\Omega(p,t)$ is forever the same correct process for all processes (leader election)
- ◇ $S$: completeness + eventual weak accuracy

$\Omega \geq \langle \rangle S$: clear
$\langle \rangle S \geq \Omega$:
  each process $p : V[p] :$ array of size $n$
  (code of $p$)
  each $k$ steps:
    $Q =$ query the failure detector $\langle \rangle S$
    for all processes $q :$ if $q$ in $Q$ then $V[p][q]$ is incremented
    compute $V[p][q] = \max V[x][q]$
    output of $\Omega = q$ such that $V[p][q]$ is the smallest element of $V[p][\ast]$
quorum failure detector $\Sigma$

- **Majority**: ensures the quorum property i.e. given any two sets of messages received from a majority of processes at least one comes from the same process

- $\Sigma$ quorum failure detector.
  
  history $H \in \Sigma$: $H(p,t)$ list of processes (trusted)

  - **intersection**: every two lists intersect
    for all $p, q$ for all $t, t'$ $H(p,t) \cap H(q,t')$

  - **completeness**: there is some time after which for all $p$
    $H(p,t) \subseteq \text{Correct}(F)$
Majority and $\Sigma$

With a majority of correct prowesses:

Output=$I$ (set of all processes)

repeat forever
send(ARE_YOU_ALIVE,r) to all
wait until receive (I_AM_ALIVE,r) from a majority $\Sigma=${\{q| a message (I_AM_ALIVE,r) received from q\}}
$r:=r+1$
||
when receive (ARE_YOU_ALIVE,r) from q
send (I_AM_ALIVE,r) to q

remark: if a majority of processes are correct then the « wait » always terminate (and all correct processes progress)

completeness: let $t$ s.t. after time $t$ no new process crashes and all messages from crashed processes are received; $p$ receives only messages from processes that are alive

intersection: two majorities intersect!

with a majority of correct processes $\Sigma$ is implementable
\[ \Sigma \text{ weakest FD for registers} \]

We have to prove:

1. With \( \Sigma \) it is possible to implement registers.

   done!

2. Consider an implementation of registers with a failure detector \( \mathcal{D} \), we have to prove that \( \Sigma \preceq \mathcal{D} \) (we can extract \( \Sigma \) from \( \mathcal{D} \))
\( \Sigma \) is the weakest failure detector to implement registers in message passing
Consensus = $\Omega + CA$

**Shared:**
- $D[1,\ldots,\infty]$, regular registers, initially $T$
- $CA_1, CA_2, \ldots$ a series of commit-adopt instances

**Upon propose(v) by process $p_i$:**

$v_i := v$
$r := 0$

repeat forever
  $r++$
  $(c,v_i):=CA_r(v_i)$  // $r$-th instance of commit-adopt
  if $c=true$ then
    $D[r]:=v_i$  // let the others learn your value
    return $v_i$
  repeat
    if $\Omega$ outputs $p_i$ then
      $D[r]:=v_i$  // advertise your value if leader
    until $D[r]=v'$ where $v'\neq T$  // wait until the leader writes its value
  $v_i := v'$  // adopt the leader’s value

$\Omega$ enables to solve consensus
Weakest failure detector to solve consensus

- failure detector $\Omega$: outputs at each correct process a leader (process id) such that eventually all correct processes agree forever in the same leader and this leader is a correct process. For all histories $H$ of $\Omega$:
  $\exists$ leader $\in$ Correct(F) $\exists$ $t \quad \forall t' > t \quad \forall p \in$ Correct(F) $H(p,t') = \text{leader}$

Main result: with a majority of correct processes (or in shared memory) $\Omega$ is the weakest failure detector to solve the consensus [CHT16]

Extension: $\Omega \times \Sigma$ is the weakest failure detector to solve the consensus in message passing [DFG10]

to get consensus we need a leader!
Solving problem P
(practical point of view)

- Given some problem P that is not solvable in asynchronous models

- From a problem P, find the weakest failure detector WFD(P) for P

- Find the « best » model in which WFD(P) can be implemented + a « good » implementation of WFD(P)

- With the « good » implementation of WFD(P) and a (good) algorithm using FD: WFD(P) solves P
Implementing failure detectors

• Implementing $\Omega$:

  • « good » partially synchronous model:

    • link p to q is *eventually timely*: there exists delta, there exists a time after which all messages from p to q are received in less than delta

    • if there is a (correct) process p such that all links from p are eventually timely then $\Omega$ may be implemented (source)

    • (then in such systems consensus is possible)
implementing $\Omega$

process $p$:
- sends regularly message(alive) to all

when no (alive) message from $q$ since $\text{timeout}_p[q]$
  - $\text{timeout}_p[q] = \text{timeout}_p[q] + 1$
  - $\text{counter}_p[q] = \text{counter}_p[q] + 1$
- reset timer for alive message from $q$
- send to all ($\text{counter}_p$)

when receives (counter)
  - $\text{counter}_p = \text{max}(\text{counter}_p, \text{counter})$
  - send to all ($\text{counter}_p$)

leader$_p = \text{min}\{r | \text{counter}_p(r) = \text{min}\{\text{counter}_p(x) | x \in \mathcal{I}\}\}$

Consider a source $s$:
- at some time for all (alive) processes $\text{timeout}_p[s]$ is greater than communication delay from $s$
- then $\text{counter}_p[s]$ stops increasing
- eventually
  - $\text{counter}_p[s] = \text{counter}_q[s]$

$q$ is not a source $\text{counter}_p[q]$ increases forever for all $p$
Conclusion

• ...