Chapter 14: Consensus and Agreement

Ajay Kshemkalyani and Mukesh Singhal

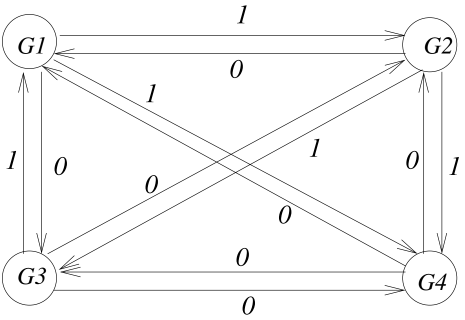
Distributed Computing: Principles, Algorithms, and Systems

Cambridge University Press

Assumptions

System assumptions

- Failure models
- Synchronous/ Asynchronous communication
- Network connectivity
- Sender identification
- Channel reliability
- Authenticated vs. non-authenticated messages
- Agreement variable



Problem Specifications

Byzantine Agreement (single source has an initial value)

- **Agreement:** All non-faulty processes must agree on the same value.
- **Validity:** If the source process is non-faulty, then the agreed upon value by all the non-faulty processes must be the same as the initial value of the source.
- **Termination:** Each non-faulty process must eventually decide on a value.

Consensus Problem (all processes have an initial value)

- **Agreement:** All non-faulty processes must agree on the same (single) value.
- **Validity:** If all the non-faulty processes have the same initial value, then the agreed upon value by all the non-faulty processes must be that same value.
- **Termination:** Each non-faulty process must eventually decide on a value.

Interactive Consistency (all processes have an initial value)

- **Agreement:** All non-faulty processes must agree on the same array of values $A[v_1 \ldots v_n]$.
- **Validity:** If process i is non-faulty and its initial value is v_i, then all non-faulty processes agree on v_i as the ith element of the array A. If process j is faulty, then the non-faulty processes can agree on any value for $A[j]$.
- **Termination:** Each non-faulty process must eventually decide on the array A.

These problems are equivalent to one another! Show using reductions.

Overview of Results

Failure mode	Synchronous system (message-passing and shared memory)	Asynchronous system (message-passing and shared memory)
No failure	agreement attainable; common knowledge also attainable	agreement attainable; concurrent common knowledge attainable
Crash failure	agreement attainable $f < n$ processes $\Omega(f+1)$ rounds	agreement not attainable
Byzantine failure	agreement attainable $f \leq \lfloor (n-1)/3 \rfloor$ Byzantine processes $\Omega(f+1)$ rounds	agreement not attainable

Table: Overview of results on agreement. f denotes number of failure-prone processes. n is the total number of processes.

In a failure-free system, consensus can be attained in a straightforward manner

Some Solvable Variants of the Consensus Problem in Async Systems

Solvable Variants	Failure model and overhead	Definition
Reliable broadcast	crash failures, $n > f$ (MP)	Validity, Agreement, Integrity conditions
k-set consensus	crash failures. $f < k < n$. (MP and SM)	size of the set of values agreed upon must be less than k
ϵ-agreement	crash failures $n \geq 5f + 1$ (MP)	values agreed upon are within ϵ of each other
Renaming	up to f fail-stop processes, $n \geq 2f + 1$ (MP) Crash failures $f \leq n - 1$ (SM)	select a unique name from a set of names

Table: Some solvable variants of the agreement problem in asynchronous system. The overhead bounds are for the given algorithms, and not necessarily tight bounds for the problem.

Solvable Variants of the Consensus Problem in Async Systems

Circumventing the impossibility results for consensus in asynchronous systems

Message−passing

k **set consensus**

epsilon− **consensus**

Renaming

Reliable broadcast

Shared memory

k **set consensus**

epsilon− **consensus**

Renaming

using atomic registers and atomic snapshot objects constructed from atomic registers

Consensus

using more powerful objects than atomic registers. This is the study of universal objects and universal constructions.

Consensus Algorithm for Crash Failures (MP, synchronous)

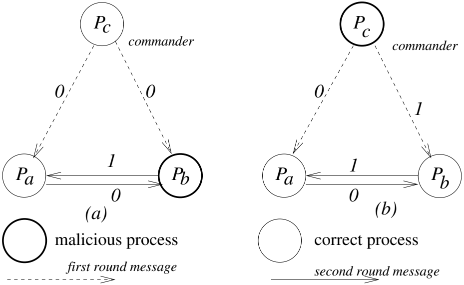
- Up to f $(< n)$ crash failures possible.
- In $f + 1$ rounds, at least one round has no failures.
- Now justify: agreement, validity, termination conditions are satisfied.
- Complexity: $O(f + 1)n^2$ messages
- $f + 1$ is lower bound on number of rounds

(global constants)
integer: f; // maximum number of crash failures tolerated
(local variables)
integer: $x \longleftarrow$ local value;

(1) Process P_i $(1 \leq i \leq n)$ executes the Consensus algorithm for up to f crash failures:
(1a) **for** *round* **from** 1 **to** $f + 1$ **do**
(1b)   **if** the current value of x has not been broadcast **then**
(1c)   **broadcast**(x);
(1d) $y_j \longleftarrow$ value (if any) received from process j in this round;
(1e) $x \longleftarrow min(x, y_j)$;
(1f) **output** x as the consensus value.

Upper Bound on Byzantine Processes (sync)

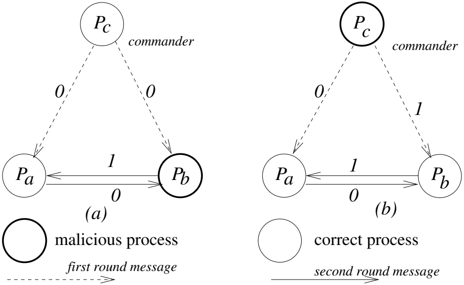
Agreement impossible when $f = 1, n = 3$.



- Taking simple majority decision does not help because loyal commander P_a cannot distinguish between the possible scenarios (a) and (b);
- hence does not know which action to take.
- Proof using induction that problem solvable if $f \leq \lfloor \frac{n-1}{3} \rfloor$. See text.

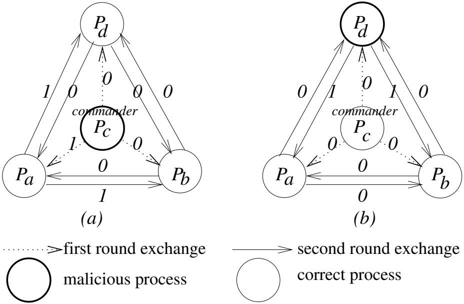
Upper Bound on Byzantine Processes (sync)

Agreement impossible when $f = 1, n = 3$.



- Taking simple majority decision does not help because loyal commander P_a cannot distinguish between the possible scenarios (a) and (b);
- hence does not know which action to take.
- Proof using induction that problem solvable if $f \leq \lfloor \frac{n-1}{3} \rfloor$. See text.

Consensus Solvable when $f = 1, n = 4$



- There is no ambiguity at any loyal commander, when taking majority decision
- Majority decision is over 2nd round messages, and 1st round message received directly from commander-in-chief process.

Byzantine Generals (recursive formulation), (sync, msg-passing)

(variables)
boolean: $v \longleftarrow$ initial value;
integer: $f \longleftarrow$ maximum number of malicious processes, $\leq \lfloor (n-1)/3 \rfloor$;
(message type)
$Oral\_Msg(v, Dests, List, faulty)$, where
v is a boolean,
Dests is a set of destination process ids to which the message is sent,
List is a list of process ids traversed by this message, ordered from most recent to earliest,
faulty is an integer indicating the number of malicious processes to be tolerated.

Oral_Msg(f), where $f > 0$:

1. The algorithm is initiated by the Commander, who sends his source value v to all other processes using a $OM(v, N, \langle i \rangle, f)$ message. The commander returns his own value v and terminates.
2. **[Recursion unfolding:]** For each message of the form $OM(v_j, Dests, List, f')$ received in this round from some process j, the process i uses the value v_j it receives from the source, and using that value, acts as a *new* source. (If no value is received, a default value is assumed.)
   To act as a new source, the process i initiates $Oral\_Msg(f'-1)$, wherein it sends
   $OM(v_j, Dests - \{i\}, concat(\langle i \rangle, L), (f'-1))$
   to destinations not in $concat(\langle i \rangle, L)$
   in the next round.
3. **[Recursion folding:]** For each message of the form $OM(v_j, Dests, List, f')$ received in Step 2, each process i has computed the agreement value v_k, for each k not in *List* and $k \neq i$,corresponding to the value received from P_k after traversing the nodes in *List*, at one level lower in the recursion. If it receives no value in this round, it uses a default value. Process i then uses the value $majority_{k \notin List, k \neq i}(v_j, v_k)$ as the agreement value and returns it to the next higher level in the recursive invocation.

Oral_Msg(0):

1. **[Recursion unfolding:]** Process acts as a source and sends its value to each other process.
2. **[Recursion folding:]** Each process uses the value it receives from the other sources, and uses that value as the agreement value. If no value is received, a default value is assumed.

Relationship between # Messages and Rounds

round number	a message has already visited	aims to tolerate these many failures	and each message gets sent to	total number of messages in round
1	1	f	$n-1$	$n-1$
2	2	$f-1$	$n-2$	$(n-1)\cdot(n-2)$
...	...	...	...	...
x	x	$(f+1)-x$	$n-x$	$(n-1)(n-2)\ldots(n-x)$
$x+1$	$x+1$	$(f+1)-x-1$	$n-x-1$	$(n-1)(n-2)\ldots(n-x-1)$
$f+1$	$f+1$	0	$n-f-1$	$(n-1)(n-2)\ldots(n-f-1)$

Table: Relationships between messages and rounds in the Oral Messages algorithm for Byzantine agreement.

Complexity: $f+1$ rounds, exponential amount of space, and

$$(n-1)+(n-1)(n-2)+\ldots+(n-1)(n-2)..(n-f-1)\text{messages}$$

Bzantine Generals (iterative formulation), Sync, Msg-passing

(variables)
boolean: $v \longleftarrow$ initial value;
integer: $f \longleftarrow$ maximum number of malicious processes, $\leq \lfloor \frac{n-1}{3} \rfloor$;
tree of boolean:

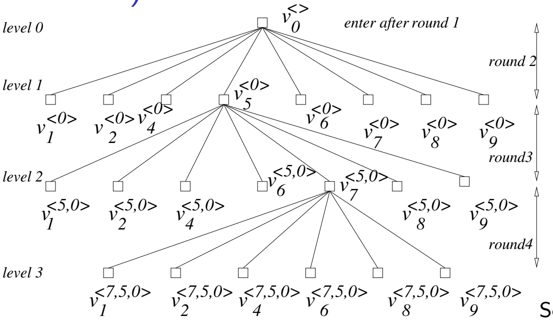
- level 0 root is v^L_{init}, where $L = \langle\rangle$;
- level $h(f \geq h > 0)$ nodes: for each v^L_j at level $h - 1 = sizeof(L)$, its $n - 2 - sizeof(L)$ descendants at level h are $v_k^{concat(\langle j \rangle, L)}$, $\forall k$ such that $k \neq j, i$ and k is not a member of list L.

(message type)
OM(v, $Dests$, $List$, $faulty$), where the parameters are as in the recursive formulation.

(1) Initiator (i.e., Commander) initiates Oral Byzantine agreement:
(1a) **send** $OM(v, N - \{i\}, \langle P_i \rangle, f)$ to $N - \{i\}$;
(1b) **return**(v).

(2) (Non-initiator, i.e., Lieutenant) receives Oral Message OM:
(2a) **for** $rnd = 0$ **to** f **do**
(2b) **for** each message OM that arrives in this round, **do**
(2c) **receive** $OM(v, Dests, L = \langle P_{k_1} \ldots P_{k_{f+1-faulty}} \rangle, faulty)$ from P_{k_1};
// $faulty + round = f$; $|Dests| + sizeof(L) = n$
(2d) $v^{tail(L)}_{head(L)} \longleftarrow v$; // $sizeof(L) + faulty = f + 1$. fill in estimate.
(2e) **send** $OM(v, Dests - \{i\}, \langle P_i, P_{k_1} \ldots P_{k_{f+1-faulty}} \rangle, faulty - 1)$ to $Dests - \{i\}$ **if** $rnd < f$;
(2f) **for** $level = f - 1$ **down to** 0 **do**
(2g) **for** each of the $1 \cdot (n - 2) \cdot \ldots (n - (level + 1))$ nodes v^L_x in level $level$, **do**
(2h) $v^L_x(x \neq i, x \notin L) = majority_{y \notin concat(\langle x \rangle, L); y \neq i}(v^L_x, v_y^{concat(\langle x \rangle, L)})$;

Tree Data Structure for Agreement Problem (Byzantine Generals)



Some branches of the tree at P_3. In this example, $n = 10, f = 3$, commander is P_0.

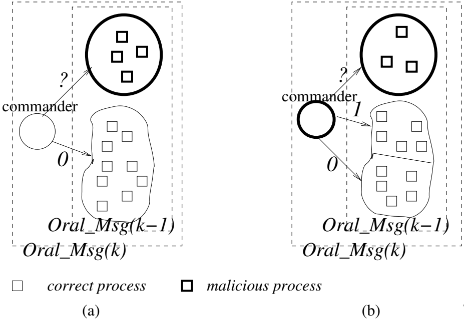
- (round 1) P_0 sends its value to all other processes using *Oral_Msg*(3), including to P_3.
- (round 2) P_3 sends 8 messages to others (excl. P_0 and P_3) using *Oral_Msg*(2). P_3 also receives 8 messages.
- (round 3) P_3 sends $8 \times 7 = 56$ messages to all others using *Oral_Msg*(1); P_3 also receives 56 messages.
- (round 4) P_3 sends $56 \times 6 = 336$ messages to all others using *Oral_Msg*(0); P_3 also receives 336 messages. The received values are used as estimates of the majority function at this level of recursion.

Exponential Algorithm: An example

An example of the majority computation is as follows.

- P_3 revises its estimate of $v_7^{\langle 5,0 \rangle}$ by taking $majority(v_7^{\langle 5,0 \rangle}, v_1^{\langle 7,5,0 \rangle}, v_2^{\langle 7,5,0 \rangle}, v_4^{\langle 7,5,0 \rangle}, v_6^{\langle 7,5,0 \rangle}, v_8^{\langle 7,5,0 \rangle}, v_9^{\langle 7,5,0 \rangle})$. Similarly for the other nodes at level 2 of the tree.
- P_3 revises its estimate of $v_5^{\langle 0 \rangle}$ by taking $majority(v_5^{\langle 0 \rangle}, v_1^{\langle 5,0 \rangle}, v_2^{\langle 5,0 \rangle}, v_4^{\langle 5,0 \rangle}, v_6^{\langle 5,0 \rangle}, v_7^{\langle 5,0 \rangle}, v_8^{\langle 5,0 \rangle}, v_9^{\langle 5,0 \rangle})$. Similarly for the other nodes at level 1 of the tree.
- P_3 revises its estimate of $v_0^{\langle \rangle}$ by taking $majority(v_0^{\langle \rangle}, v_1^{\langle 0 \rangle}, v_2^{\langle 0 \rangle}, v_4^{\langle 0 \rangle}, v_5^{\langle 0 \rangle}, v_6^{\langle 0 \rangle}, v_7^{\langle 0 \rangle}, v_8^{\langle 0 \rangle}, v_9^{\langle 0 \rangle})$. This is the consensus value.

Impact of a Loyal and of a Disloyal Commander



The effects of a loyal or a disloyal commander in a system with $n = 14$ and $f = 4$. The subsystems that need to tolerate k and $k-1$ traitors are shown for two cases. (a) Loyal commander. (b) No assumptions about commander.

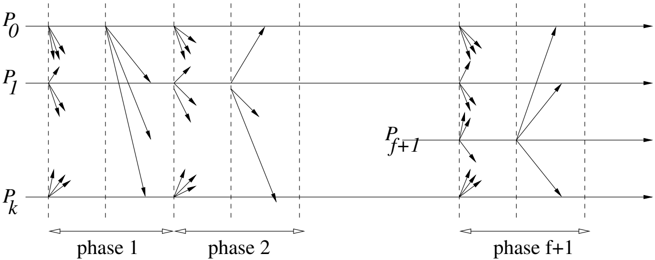
(a) the commander who invokes *Oral_Msg(x)* is loyal, so all the loyal processes have the same estimate. Although the subsystem of $3x$ processes has x malicious processes, all the loyal processes have the same view to begin with. Even if this case repeats for each nested invocation of *Oral_Msg*, even after x rounds, among the processes, the loyal processes are in a simple majority, so the majority function works in having them maintain the same common view of the loyal commander's value.

(b) the commander who invokes *Oral_Msg(x)* may be malicious and can send conflicting values to the loyal processes. The subsystem of $3x$ processes has $x-1$ malicious processes, but all the loyal processes do not have the same view to begin with.

The Phase King Algorithm

Operation

- Each phase has a unique "phase king" derived, say, from PID.
- Each phase has two rounds:
  1. in 1st round, each process sends its estimate to all other processes.
  2. in 2nd round, the "Phase king" process arrives at an estimate based on the values it received in 1st round, and broadcasts its new estimate to all others.



The Phase King Algorithm: Code

(variables)
boolean: $v \longleftarrow$ initial value;
integer: $f \longleftarrow$ maximum number of malicious processes, $f < \lceil n/4 \rceil$;

(1) Each process executes the following $f+1$ phases, where $f < n/4$:
(1a) **for** *phase* $= 1$ **to** $f+1$ **do**
(1b) Execute the following Round 1 actions: // actions in round one of each phase
(1c) **broadcast** v to all processes;
(1d) **await** value v_j from each process P_j;
(1e) *majority* $\longleftarrow$ the value among the v_j that occurs $> n/2$ times (default if no maj.);
(1f) *mult* $\longleftarrow$ number of times that *majority* occurs;
(1g) Execute the following Round 2 actions: // actions in round two of each phase
(1h) **if** $i =$ *phase* **then** // only the phase leader executes this send step
(1i) **broadcast** *majority* to all processes;
(1j) **receive** *tiebreaker* from P_{phase} (default value if nothing is received);
(1k) **if** *mult* $> n/2 + f$ **then**
(1l) $v \longleftarrow$ *majority*;
(1m) **else** $v \longleftarrow$ *tiebreaker*;
(1n) **if** *phase* $= f+1$ **then**
(1o) output decision value v.

The Phase King Algorithm

- $(f+1)$ phases, $(f+1)[(n-1)(n+1)]$ messages, and can tolerate up to $f < \lceil n/4 \rceil$ malicious processes

Correctness Argument

1. Among $f+1$ phases, at least one phase k where phase-king is non-malicious.
2. In phase k, all non-malicious processes P_i and P_j will have same estimate of consensus value as P_k does.
   1. P_i and P_j use their own majority values (Hint: $\implies P_i$'s $mult > n/2 + f$)
   2. P_i uses its majority value; P_j uses phase-king's tie-breaker value. (Hint: P_i"s $mult > n/2 + f$, P_j's $mult > n/2$ for same value)
   3. P_i and P_j use the phase-king's tie-breaker value. (Hint: In the phase in which P_k is non-malicious, it sends same value to P_i and P_j)

   In all 3 cases, argue that P_i and P_j end up with same value as estimate
3. If all non-malicious processes have the value x at the start of a phase, they will continue to have x as the consensus value at the end of the phase.

Impossibility Result (MP, async)

FLP Impossibility result

Impossible to reach consensus in an async MP system even if a single process has a crash failure

- In a failure-free async MP system, initial state is *monovalent* $\Longrightarrow$ consensus can be reached.
- In the face of failures, initial state is necessarily bivalent
- Transforming the input assignments from the all-0 case to the all-1 case, there must exist input assignments $\vec{I_a}$ and $\vec{I_b}$ that are 0-valent and 1-valent, resp., and that differ in the input value of only one process, say P_i. If a 1-failure tolerant consensus protocol exists, then:
  - Starting from $\vec{I_a}$, if P_i fails immediately, the other processes must agree on 0 due to the termination condition.
  - Starting from $\vec{I_b}$, if P_i fails immediately, the other processes must agree on 1 due to the termination condition.

  However, execution (2) looks identical to execution (1), to all processes, and must end with a consensus value of 0, a contradiction. Hence, there must exist at least one bivalent initial state.
- Consensus requires some communication of initial values.
- Key idea: in the face of a potential crash, not possible to distinguish between

Impossibility Result (MP, async)

- To transition from bivalent to monovalent step, must exist a critical step which allows the transition by making a decision
- Critical step cannot be local (cannot tell apart between slow and failed process) nor can it be across multiple processes (it would not be well-defined)
- Hence, cannot transit from bivalent to univalent state.

Wider Significance of Impossibility Result

- By showing reduction from consensus to problem X, then X is also not solvable under same model (single crash failure)
- E.g., leader election, terminating reliable broadcast, atomic broadcast, computing a network-wide global function using BC-CC flows, transaction commit.

Terminating Reliable Broadcast (TRB)

A correct process always gets a message, even if sender crashes while sending (in which case the process gets a null message).

Validity: If the sender of a broadcast message m is non-faulty, then all correct processes eventually deliver m.

Agreement: If a correct process delivers a message m, then all correct processes deliver m.

Integrity: Each correct process delivers at most one message. Further, if it delivers a message different from the null message, then the sender must have broadcast m.

Termination: Every correct process eventually delivers some message.

Reduction from consensus to TRB.

- Commander sends its value using TRB.
- Receiver decides on 0 or 1 based on value it receives. If it receives a "null" message, it decides on default value.
- But, as consensus is not solvable, algo for TRB cannot exist.

k-set Consensus

- k-Agreement: All non-faulty processes must make a decision, and the set of values that the processes decide on can contain up to k ($> f$) values.
- Validity: If a non-faulty process decides on some value, then that value must have been proposed by some process.
- Termination: Each non-faulty process must eventually decide on a value.

The k-Agreement condition is new, the Validity condition is different from that for regular consensus, and the Termination condition is unchanged from that for regular consensus.

Example: Let $n = 10, f = 2, k = 3$ and each process choose a unique number from 1 to 10. Then 3-set is $\{8, 9, 10\}$.

(variables)
integer: $v \longleftarrow$ initial value;

(1) A process P_i, $1 \leq i \leq n$, initiates k-set consensus:
(1a) **broadcast** v to all processes.
(1b) **await** values from $|N| - f$ processes and add them to set V;
(1c) **decide** on $max(V)$.

Epsilon Consensus (msg-passing, async)

ϵ-Agreement: All non-faulty processes must make a decision and the values decided upon by any two non-faulty processes must be within ϵ range of each other.

Validity: If a non-faulty process P_i decides on some value v_i, then that value must be within the range of values initially proposed by the processes.

Termination: Each non-faulty process must eventually decide on a value.

The algorithm for the message-passing model assumes $n \geq 5f + 1$, although the problem is solvable for $n > 3f + 1$.

- Main loop simulates sync rounds.
- Main lines (1d)-(1f): processes perform all-all msg exchange
- Process broadcasts its estimate of consensus value, and awaits $n - f$ similar msgs from other processes
- the processes' estimate of the consensus value converges at a particular rate, until it is ϵ from any other processes estimate.
- # rounds determined by lines (1a)-(1c).

Epsilon Consensus (msg-passing, async): Code

(variables)
real: $v \longleftarrow$ input value; //initial value
multiset of real V;
integer $r \longleftarrow 0$; // number of rounds to execute

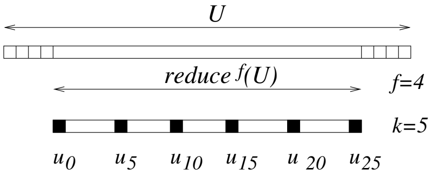
(1) Execution at process $P_i, 1 \leq i \leq n$:
(1a) $V \longleftarrow Asynchronous\_Exchange(v, 0)$;
(1b) $v \longleftarrow$ any element in$(reduce^{2f}(V))$;
(1c) $r \longleftarrow \lceil log_c(diff(V))/\epsilon \rceil$, where $c = c(n-3f, 2f)$.
(1d) **for** *round* **from** 1 **to** r **do**
(1e) $V \longleftarrow Asynchronous\_Exchange(v, round)$;
(1f) $v \longleftarrow new_{2f,f}(V)$;
(1g) **broadcast** $(\langle v, \textbf{halt} \rangle, r+1)$;
(1h) **output** v as decision value.

(2) *Asynchronous_Exchange(v,h)* returns V:
(2a) **broadcast** (v, h) to all processes;
(2b) **await** $n - f$ responses belonging to round h;
(2c) for each process P_k that sent $\langle x, \textbf{halt} \rangle$ as value, use x as its input henceforth;
(2d) **return** the multiset V.

Epsilon Consensus (msg-passing, async)

Consider a sorted collection U. The new estimate of a process is chosen by computing $new_{k,f}(U)$, defined as $mean(select_k(reduce^f(U)))$

- $reduce^f(U)$ removes the f largest and f smallest members of U.
- $select_k(U)$ selects every kth member of U, beginning with the first. If U has m members, $select_k(U)$ has $c(m,k) = \lfloor (m-1)/k \rfloor + 1$ members. This constant c represents a *convergence factor* towards the final agreement value, i.e., if x is the range of possible values held by correct processes before a round, then x/c is the possible range of estimate values held by those processes after that round.



shaded members belong to $select_5(reduce^4(U))$

$select_k(reduce^f(U))$ operation, with $k=5$ and $f=4$. The mean of the selected members is the new estimate $new_{5,4}(U)$.

The algorithm uses $m = n - 3f$ and $k = 2f$. So $c(n-3f, 2f)$ will represent the *convergence factor* towards reaching approximate agreement and $new_{2f,f}$ is the new estimate after each round.

Epsilon Consensus (msg-passing, async)

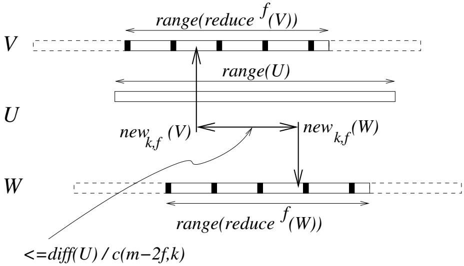
Let $|U| = m$, and let the m elements $u_0 \ldots u_{m-1}$ of multiset U be in nondecreasing order.
Properties on nonempty multisets U, V, W.

- The number of the elements in multisets U and V is reduced by at most 1 when the smallest element is removed from both. Similarly for the largest element.
- The number of elements common to U and V before and after j reductions differ by at most $2j$. Thus, for $j \geq 0$ and $|V|, |W| \geq 2j$, $|V \cap W| - |reduce^j(V) \cap reduce^j(W)| \leq 2j$.
- Let V contain at most j values not in U, i.e., $|V - U| \leq j$, and let size of V be at least $2j$. Then by removing the j low and j high elements from V, it is easy to see that remaining elements in V must belong to the range of U.

Thus,

- each value in $reduce^j(V)$ is in the range of U, i.e., $range(reduce^j(V)) \subseteq range(U)$.
- $new_{k,j}(V) \in range(U)$.

Correctness, termination, complexity: refer book



Asynchronous Renaming

The renaming problem assigns to each process P_i, a name m_i from a domain M, and is formally specified as follows.

- **Agreement:** For non-faulty processes P_i and P_j, $m_i \neq m_j$.
- **Termination:** Each non-faulty process is eventually assigned a name m_i.
- **Validity:** The name m_i belongs to M.
- **Anonymity:** The code executed by any process must not depend on its initial identifier.

Uses of renaming (name space transformation):

- processes from different domains need to collaborate, but must first assign themselves distinct names from a small domain.
- processes need to use their names as "tags" to simply mark their presence, as in a priority queue.
- the name space has to be condensed, e.g., for k-mutex.

Assumptions

- The n processes $P_1 \ldots P_n$ have their identifiers in the old name space. P_i knows only its identifier, and the total number of processes, n.
- The n processes take on new identifiers $m_1 \ldots m_n$, resp., from the name space M.
- Due to asynchrony, each process that chooses its new name must continue to cooperate with the others until they have chosen their new names.

Asynchronous Renaming -MP Model

- Attiya et al. renaming algorithm assumes $n \geq 2f + 1$ and fail-stop model.
- Transformed name space is $M = n + f$.
- *View* is a list of up to n objects of type *bid*.

(local variables)
struct *bid*:
  integer P; // old name of process
  integer x; // new name being bid by the process
  integer *attempt*; // the number of bids so far, including this current bid
  boolean *decide*; // whether new name x is finalized
list of bid: $View[1 \ldots n] \longleftarrow \langle\langle i, 0, 0, false\rangle\rangle$; // initialize list with an entry for P_i
integer *count*; // number of copies of the latest local view, received from others
boolean: *restart*, *stable*, *no_choose*; // loop control variables

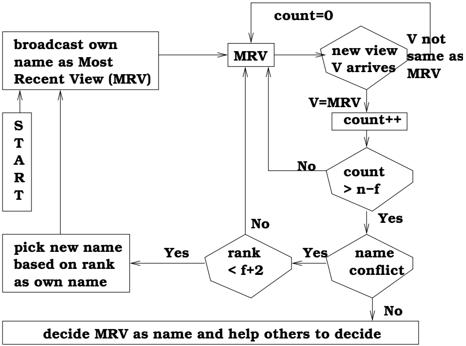
- $View \leq View'$ if and only if for each process P_i such that $View[k].P = P_i$, we also have that for some k', $View'[k'].P = P_i$ and $View[k].attempt \leq View'[k'].attempt$.

If $View' \not\leq View$ (line 1n), then *View* is updated using $View'$ (line 1o) by:

1. including all process entries from $View'$ that are missing in *View* (i.e., $View'[k'].P$ is not equal to $View[k].P$, for all k), so such entries $View'[k']$ are added to *View*.
2. replacing older entries for the same process with more recent ones, (i.e., if $View'[k'].P = P_i = View[k].P$ and $View'[k'].attempt > View[k].attempt$, replace $View[k]$ by $View'[k']$).

Asynchronous Renaming

- The high level functioning is given by the flow-chart.
- A view becomes stable if it gets $n - f$ votes.
- If no name conflict, it decides on its view and helps other processes to reach their view.
- If name conflict, it decides whether to seek more votes or try to get a new name, based on its *rank*, which is like a sequence number determined from the old name space, from among those processes who have not yet finalized their new names.
- Safety, Liveness, Termination, Complexity: refer book



Wait-free Renaming: Code

(1) A process P_i, $1 \leq i \leq n$, participates in renaming:

```
repeat
  restart ←− false;
  broadcast message(View);
  count ←− 1;
  repeat
      no_choose ←− 0;
      repeat
          await message(View′);
          stable ←− false;
          if View′ = View then
              count ←− count + 1;
              if count ≥ n − f then
                  stable ←− true;
          else if View′ ≰ View then
              update View using View′ by taking latest information for each process;
              restart ←− true;
      until (stable = true or restart = true);      // n − f copies received, or new view obtained
      if restart = false then      // View[1] has information about P_i
          if View[1].x ≠ 0 and View[1].x ≠ View[j].x for any j then
              decide View[1].x;
              View[1].decide ←− true;
              broadcast message(View);
          else
              let r be the rank of P_i in UNDECIDED(View);
              if r ≤ f + 1 then
                  View[1].x ←− FREE(View)(r), the rth free name in View;
                  View[1].attempt ←− View[1].attempt + 1;
                  restart ←− 1;
              else
                  no_choose ←− 1;
  until no_choose = 0;
until restart = 0;
repeat
  on receiving message(View′)
      update View with View′ if necessary;
      broadcast message(View);
until false.
```

Reliable Broadcast

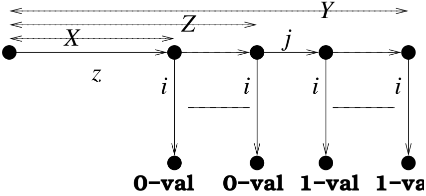
- Reliable Broadcast is RTB without terminating condition.
- RTB requires eventual delivery of messages, even if sender fails before sending. In this case, a null message needs to get sent. In RB, this condition is not there.
- RTB requires recognition of a failure, even if no msg is sent
- Crux: RTB is required to distinguish between a failed process and a slow process.
- RB is solvable under crash failures; $O(n^2)$ messages

(1) Process P_0 initiates Reliable Broadcast:
(1a) **broadcast** message M to all processes.

(2) A process P_i, $1 \leq i \leq n$, receives message M:
(2a) **if** M was not received earlier **then**
(2b) **broadcast** M to all processes;
(2c) deliver M to the application.

Shared Memory Consensus (async): Impossibility

- Use FLP argument seen in async MP systems here for SM systems.
- Cannot distinguish between failed process and a slow process $\Longrightarrow$ consensus not possible.
- Proof by contradiction, using notion of *critical step* at which system transitions from bivalent to monovalent state.
- Given initial bivalent state, prefix Z, then step by P_i leads to 0-valent state but event at some P_j followed by step of P_i leads to 1-valent state.
- Apply case analysis on prefix Z and actions of P_i and P_j after Z.



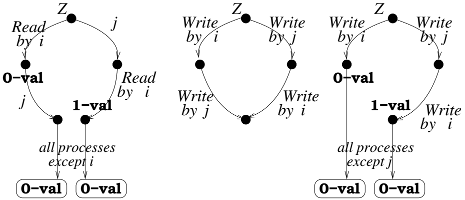
Shared Memory Consensus (async): Impossibility

(a) P_i does a Read. $extend(Z, i \cdot j)$ and $extend(Z, j \cdot i)$ are isomorphic to all except P_i. If P_i stops after $extend(Z, i \cdot j)$, all must reach consensus 0 after some suffix δ. However, as per Figure (a), processes must reach consensus 1 after δ. A contradiction.

(a') P_j does a Read. Similar reasoning to case (a)

(b) P_i and P_j Write to different vars. System state after $extend(Z, i \cdot j)$ and $extend(Z, j \cdot i)$ will have to be 0-valent and 1-valent, resp.. A contradiction.

(c) P_i and P_j Write to the same variable. System states after $extend(Z, i)$ and after $extend(Z, j \cdot i)$ are isomorphic to all except P_j. Assume P_j does not run now. Then a contradiction can be seen, because of consensus value 0 after the first prefix and a consensus value of 1 after the second prefix.



(a) i does a Read (same logic if j does a Read)

(b) i and j write to different variables

(c) i and j write to the same variable

Wait-free SM Consensus using Shared Objects

Not possible to go from bivalent to univalent state if even a single failure is allowed. Difficulty is not being able to read & write a variable atomically.

- It is not possible to reach consensus in an asynchronous shared memory system using Read/Write atomic registers, even if a single process can fail by crashing.
- There is no wait-free consensus algorithm for reaching consensus in an asynchronous shared memory system using Read/Write atomic registers.

To overcome these negative results

- Weakening the consensus problem, e.g., k-set consensus, approximate consensus, and renaming using atomic registers.
- Using memory that is stronger than atomic Read/Write memory to design wait-free consensus algorithms. Such a memory would need corresponding access primitives.

Stronger objects?

Are there objects (with supporting operations), using which there is a wait-free (i.e., $(n-1)$-crash resilient) algorithm for reaching consensus in a n-process system? Yes, e.g., Test&Set, Swap, Compare&Swap.

Henceforth, assume only the crash failure model, and also require the solutions to be *wait-free*.

Consensus Numbers and Consensus Hierarchy

Consensus Numbers

An object of type X has consensus number k, denoted as $CN(X) = k$, if k is the largest number for which the object X can solve wait-free k-process consensus in an asynchronous system subject to $k-1$ crash failures, using only objects of type X and read/write objects.

Wait-free simulations and Consensus Numbers

For objects X and Y such that $CN(X) < CN(Y)$, there is no wait-free simulation of object Y using X and read/write registers (whose consensus number is 1) in a system with more than $CN(X)$ processes.

There does not exist any simulation of objects with $CN > 1$ using only Read/Write atomic registers $\Longrightarrow$ need stronger objects.

Object	Consensus number
Read/Write objects	1
Test-&-Set, stack, FIFO queue, Fetch-&-Inc	2
Augmented queue with `peek` - size k	k
Compare-&-Swap, Augmented queue, memory-memory move	∞
memory-memory swap, Fetch-&-Cons, store-conditional	

Definitions of Sync Operations *RMW*, Compare&Swap, Fetch&Inc

(shared variables among the processes accessing each of the different object types)
register: $Reg \longleftarrow$ initial value; // shared register initialized
(local variables)
integer: $old \longleftarrow$ initial value; // value to be returned
integer: $key \longleftarrow$ comparison value for conditional update;

(1) *RMW*(*Reg*, function f) returns *value*:
(1a) $old \longleftarrow Reg$;
(1b) $Reg \longleftarrow f(Reg)$;
(1c) **return**(*old*).

(2) *Compare&Swap*(*Reg*, *key*, *new*) returns *value*:
(2a) $old \longleftarrow Reg$;
(2b) **if** $key = old$ **then**
(2c) $\quad Reg \longleftarrow new$;
(2d) **return**(*old*).

(3) *Fetch&Inc*(*Reg*) returns *value*:
(3a) $old \longleftarrow Reg$;
(3b) $Reg \longleftarrow r + 1$;
(3c) **return**(*old*).

Two-process Wait-free Consensus using FIFO Queue

(shared variables)
queue: $Q \longleftarrow \langle 0 \rangle$; // queue Q initialized
integer: $Choice[0,1] \longleftarrow [\perp, \perp]$ // preferred value of each process
(local variables)
integer: $temp \longleftarrow 0$;
integer: $x \longleftarrow$ initial choice;

(1) Process $P_i, 0 \leq i \leq 1$, executes this for 2-process consensus using a FIFO queue:
(1a) $Choice[i] \longleftarrow x$;
(1b) $temp \longleftarrow dequeue(Q)$;
(1c) **if** $temp = 0$ **then**
(1d) **output**(x)
(1e) **else output**$(Choice[1-i])$.

Wait-free Consensus using Compare&Swap

(shared variables)
integer: $Reg \longleftarrow \perp$; // shared register *Reg* initialized
(local variables)
integer: $temp \longleftarrow 0$; // temp variable to read value of *Reg*
integer: $x \longleftarrow$ initial choice; // initial preference of process

(1) Process $P_i, (\forall i \geq 1)$, executes this for consensus using *Compare&Swap*:
(1a) $temp \longleftarrow Compare\&Swap(Reg, \perp, x)$;
(1b) **if** $temp = \perp$ **then**
(1c) **output**(x)
(1d) **else output**$(temp)$.

Read-Modify-Write (MRW) Abstraction

- RMW allows to read, and modify the register content as per some function f.
- RMW object has a CN of at least 2 because it allows the first process to access the object to leave an imprint that the object has been accessed. The other process can read the imprint.
- If the imprint can include the ID of the first process, or the choice of the first process, then $CN > 2$.
- *RMW* objects differ in their function f. A function is termed as `interfering` if for all process pairs i and j, and for all legal values v of the register,
  1. $f_i(f_j(v)) = f_j(f_i(v))$, i.e., function is commutative, or
  2. the function is not write-preserving, i.e., $f_i(f_j(v)) = f_i(v)$ or vice-versa with the roles of i and j interchanged.
- **Examples:**
  - The *Fetch&Inc* commutes even though it is write-preserving.
  - The *Test&Set* commutes and is not write-preserving.
  - The *Swap* does not commute but it is not write-preserving.

Hence, all three objects uses functions that are *interfering*.

RMW Object and Instruction

A nontrivial interfering RMW operation has consensus number $= 2$

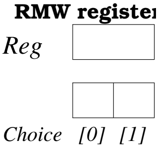
- If RMW is commutative, 3rd process cannot know which of the other two accessed the object first, and therefore does not know whose value is the consensus value
- If RMW is not write-preserving, 3rd process does not know if it is the 2nd or 3rd to access the object. Therefore, whose value is the consensus value?

Objects like Compare&Swap are non-interfering and hence have a higher consensus number.

RMW Object and Instruction

(shared variables)
integer: $Reg \longleftarrow \perp$; // shared register Reg initialized
integer: $Choice[0,1] \longleftarrow [\perp, \perp]$; // data structure
(local variables)
integer: $x \longleftarrow$ initial choice; // initial preference of process

(1) Process $P_i, (0 \leq i \leq 1)$, executes this for consensus using RMW:
(1a) $Choice[i] \longleftarrow x$;
(1b) $val \longleftarrow RMW(Reg, f)$;
(1c) **if** $val = \perp$ **then**
(1d) **output**$(Choice[i])$
(1e) **else output**$(Choice[1-i])$.



Universality of Consensus Objects

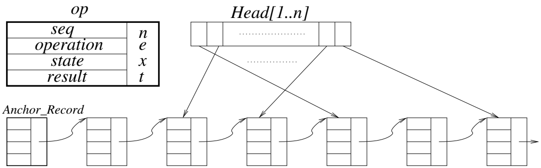
- An object is defined to be *universal* if that object along with read/write registers can simulate any other object in a wait-free manner. In any system containing up to k processes, an object X such that $CN(X) = k$ is *universal*.
- For any system with up to k processes, the universality of objects X with consensus number k is shown by giving a *universal* algorithm to wait-free simulate *any* object using only objects of type X and read/write registers. This is shown in two steps.
  1. A *universal* algorithm to wait-free simulate *any* object whatsoever using read/write registers and arbitrary k-processor consensus objects is given. This is the main step.
  2. Then, the arbitrary k-process consensus objects are simulated with objects of type X, also having consensus number k. This trivially follows after the first step.
- Hence, any object X with consensus number k is universal in a system with $n \leq k$ processes.

Universality of Consensus Objects

- An arbitrary consensus object X allows a single operation, $Decide(X, v_{in})$ and returns a value v_{out}, where both v_{in} and v_{out} have to assume a legal value from known domains V_{in} and V_{out}, resp.
- For the correctness of this shared object version of the consensus problem, all v_{out} values returned to each invoking process must equal the v_{in} of some process.
- A *nonblocking* operation, in the context of shared memory operations, is an operation that may not complete itself but is guaranteed to complete at least one of the pending operations in a finite number of steps.

A Nonblocking Universal Algorithm

- The linked list stores the linearized sequence of operations and states following each operation.
- Operations to the arbitrary object Z are simulated in a nonblocking way using only an arbitrary consensus object (namely, the field *op.next* in each record) which is accessed via the *Decide* call.
- Each process attempts to thread its own operation next into the linked list.



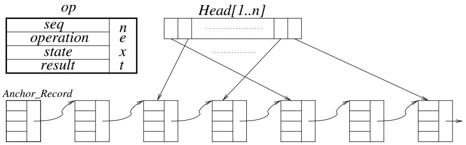
A Nonblocking Universal Algorithm

(shared variables)
record *op*
  integer: $seq \longleftarrow 0$; // sequence number of serialized operation
  $operation \longleftarrow \perp$; // operation, with associated parameters
  $state \longleftarrow initial\ state$; // the state of the object after the operation
  $result \longleftarrow \perp$; // the result of the operation, to be returned to invoker
  op $*next \longleftarrow \perp$; // pointer to the next record
op $*Head[1 \ldots k] \longleftarrow \overline{\&(anchor\_record)}$;
(local variables)
op $*my\_new\_record, *winner$;

(1) Process $P_i, 1 \leq i \leq k$ performs operation *invoc* on an arbitrary consensus object:
(1a) $my\_new\_record \longleftarrow malloc(op)$;
(1b) $my\_new\_rec.operation \longleftarrow invoc$;
(1c) **for** $count = 1$ **to** k **do**
(1d)   **if** $Head[i].seq < Head[count].seq$ **then**
(1e) $Head[i] \longleftarrow Head[count]$;
(1f) **repeat**
(1g) $winner \longleftarrow Decide(Head[i].next, \&my\_new\_record)$;
(1h) $winner.seq \longleftarrow Head[i].seq + 1$;
(1i) $winner.state, winner.result \longleftarrow apply(winner.operation, Head[i].state)$;
(1j) $Head[i] \longleftarrow winner$;
(1k) **until** $winner = my\_new\_record$;
(1l) enable the response to *invoc*, that is stored at *winner.result*.

A Nonblocking Universal Algorithm: Notes

- There are as many universal objects as there are operations to thread.
- A single pointer/counter cannot be used instead of the array *Head*. B'coz reading and updating the pointer cannot be done atomically in a wait-free manner.
- Linearization of the operations given by the seq no.
- As algorithm is nonblocking, some process(es) may be starved indefinitely.



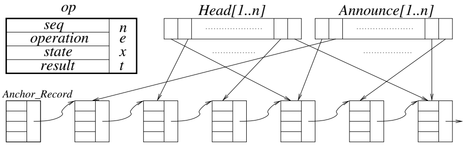
A Wait-free Universal Algorithm

(shared variables)
record *op*
    integer: $seq \longleftarrow 0$; // sequence number of serialized operation
    $operation \longleftarrow \perp$; // operation, with associated parameters
    $state \longleftarrow initial\ state$; // the state of the object after the operation
    $result \longleftarrow \perp$; // the result of the operation, to be returned to invoker
    op $*next \longleftarrow \perp$; // pointer to the next record
op $*Head[1 \ldots k], *Announce[1 \ldots k] \longleftarrow \overline{\&(anchor\_record)}$;
(local variables)
op $*my\_new\_record, *winner$;

(1) Process $P_i, 1 \leq i \leq k$ performs operation *invoc* on an arbitrary consensus object:
(1a) $Announce[i] \longleftarrow malloc(op)$;
(1b) $Announce[i].operation \longleftarrow invoc$; $Announce[i].seq \longleftarrow 0$;
(1c) **for** $count = 1$ **to** k **do**
(1d)   **if** $Head[i].seq < Head[count].seq$ **then**
(1e) $Head[i] \longleftarrow Head[count]$;
(1f) **while** $Announce[i].seq = 0$ **do**
(1g) $turn \longleftarrow (Head[i].seq + 1) mod\ (k)$;
(1h)   **if** $Announce[turn].seq = 0$ **then**
(1i) $my\_new\_record \longleftarrow Announce[turn]$;
(1j)   **else** $my\_new\_record \longleftarrow Announce[i]$;
(1k) $winner \longleftarrow Decide(Head[i].next, \&my\_new\_record)$;
(1l) $winner.seq \longleftarrow Head[i].seq + 1$;
(1m) $winner.state, winner.result \longleftarrow apply(winner.operation, Head[i].state)$;
(1n) $Head[i] \longleftarrow winner$;
(1o) enable the response to *invoc*, that is stored at $winner.result$.

Wait-free Universal Algorithm

- To prevent starvation in the nonblocking algoriithm, the idea of "helping" using a round-robin approach modulo n is used.
- If P_j determines that the next op is to be assigned sequence number x, then it first checks whether the process P_i such that $i = x\,(mod\ n)$ is contending for threading its operation. If so, then P_j tries to thread P_i's operation instead of its own.
- The round-robin approach uses the array *Announce*.
- Within n iterations of the outer loop, a process is certain that its operation gets threaded - by itself or with the help of another contending process.



Shared Memory k-set Consensus

- Crash failure model, $k > f$. Analogous to message-passing model algorithm. Assumes atomic snapshot object Obj.
- P_i writes its value to $Obj[i]$ and scans Obj until $n - f$ values have been written to it. Then takes the max.

(variables)
integer: $v \longleftarrow$ initial value;
array of integer *local_array* $\longleftarrow \overline{\perp}$;
(shared variables)
atomic snapshot object $Obj[1 \ldots n] \longleftarrow \overline{\perp}$;

(1) A process $P_i, 1 \leq i \leq n$, initiates k-set consensus:
(1a) **update**$_i$($Obj[i]$) with v;
(1b) **repeat**
(1c) *local_array* $\longleftarrow$ **scan**$_i$(Obj);
(1d) **until** there are at least $|N| - f$ non-null values in Obj;
(1e) $v \longleftarrow$ max. of the values in *local_array*.

Async Wait-free Renaming using Atomic Shared Object

- Crash failure model. *Obj* linearizes all accesses to it.
- Each P_i can write to its portion in *Obj* and read all *Obj* atomically.
- P_i does not have a unique index from $[1 \ldots n]$.
- P_i proposes a name "1" for itself. It then repeats the following loop.
  - It writes its latest bid to its component of *Obj* (line 1c); it reads the entire object using a **scan** into its local array (line 1d). P_i examines the local array for a possible conflict with its proposed new name (line 1e).
    - If P_i detects a conflict with its proposed name m_i (line 1e) it determines its rank *rank* among the *old* names (line 1f); and selects the $rank^{th}$ smallest integer among the names that have not been proposed in the view just read (line 1g). This will be used as P_i's bid for a new name in the next iteration.
    - If P_i detects no conflict with its proposed name m_i (line 1e), it selects this name and exits (line 1i).

Async Wait-free Renaming using Atomic Shared Object

Correctness: As *Obj* is linearizable, no two proceses having chosen a new name will get back a Scan saying their new names are unique.

Size of new name space: $[1 \ldots 2n - 1]$.

Termination: Assume there is a subset $\overline{T} \subseteq N$ of processes that never terminate. Let $min(\overline{T})$ be the process in $\overline{T}$ with the lowest ranked process identifier (old name). Let $rank(min(\overline{T}))$ be the rank of this process among *all* the processes $P_1 \ldots P_n$. Once every process in $\overline{T}$ has done at least one **update**, and once all the processes in T have terminated, we have the following.

- The set of names of the terminated processes, say M_T, remains fixed.
- The process $min(\overline{T})$ will choose a name not in M_T, that is ranked $rank(min(\overline{T}))$. As $rank(min(\overline{T}))$ is unique, no other process in $\overline{T}$ will ever choose this name.
- Hence, $min(\overline{T})$ will not detect any conflict with $rank(min(\overline{T}))$ and will terminate.

As $min(\overline{T})$ cannot exist, the set $\overline{T} = \emptyset$.

Lower bound: For crash-failures, lower bound of $n + f$ on new name space.

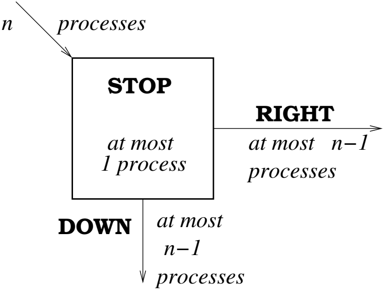
Async Wait-free Renaming using Atomic Shared Object

(variables)
integer: $m_i \longleftarrow 0$;
integer: $P_i \longleftarrow$ name from old domain space;
list of integer tuples *local_array* $\longleftarrow \overline{\langle \perp, \perp \rangle}$;
(shared variables)
atomic snapshot object *Obj* $\longleftarrow \overline{\langle \perp, \perp \rangle}$; // n components

(1) A process $P_i, 1 \leq i \leq n$, participates in wait-free renaming:
(1a) $m_i \longleftarrow 1$;
(1b) **repeat**
(1c) **update**$_i(Obj, \langle P_i, m_i \rangle)$; // update ith component with bid m_i
(1d) $local\_array(\langle P_1, m_1 \rangle, \ldots \langle P_n, m_n \rangle) \longleftarrow$ **scan**$_i(Obj)$;
(1e) **if** $m_i = m_j$ for some $j \neq i$ **then**
(1f) Determine rank $rank_i$ of P_i in $\{P_j \mid P_j \neq \perp \wedge j \in [1, n]\}$;
(1g) $m_k \longleftarrow rank_i$th smallest integer not in $\{m_j \mid m_j \neq \perp \wedge j \in [1, n] \wedge j \neq i\}$;
(1h) **else**
(1i) **decide**(m_k); **exit**;
(1j) **until** *false*.

The Splitter

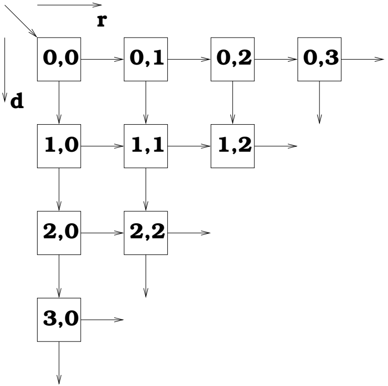
- At most one process is returned *stop*.
- At most $n-1$ processes are returned *down*.
- At most $n-1$ processes are returned *right*.



(shared variables)
MRMW atomic snapshot object X, $\longleftarrow$ 0; **MRMW atomic snapshot object** $Y \longleftarrow$ *false*;

(1) <u>*splitter()*, executed by process $P_i, 1 \leq i \leq n$:</u>
(1a) $X \longleftarrow i$;
(1b) **if** Y **then**
(1c) **return**(*right*);
(1d) **else**
(1e) $Y \longleftarrow$ *true*;
(1f) if $X = i$ **then return**(*stop*)
(1g) **else return**(*down*).

Configuration of Splitters for Wait-free Renaming (SM)



(local variables)
next, r, d, *new_name* $\longleftarrow$ 0;

(1) Process $P_i, 1 \leq i \leq n$, participates in wait-free renaming:
(1a) $r, d \longleftarrow 0$;
(1b) **while** $next_i \neq stop$ **do**
(1c) $\quad next_i \longleftarrow splitter(r, d)$;
(1d) $\quad$ **case**
(1e) $\quad\quad next = right$ **then** $r \longleftarrow r + 1$;
(1f) $\quad\quad next = down$ **then** $d \longleftarrow d + 1$;
(1g) $\quad\quad next = stop$ **then break()**
(1h) **return**$(new\_name = n \cdot d - d(d-1)/2 + r)$.

New

name space: $n(n+1)/2$ splitters