UNIT IV
CONSENSUS AND RECOVERY

Consensusand agreement algorithms: Problem definition – Overview of results – Agreement in a failure – free system (Synchronous and Asynchronous) – Agreement in synchronous systems with failures. Check pointing and rollback recovery: Introduction – Background and definitions – Issues in failure recovery – Checkpoint-based recovery – Coordinated check pointing algorithm – Algorithm for asynchronous check pointing and recovery.

CONSENSUS PROBLEM IN ASYNCHRONOUS SYSTEMS.

Table: Overview of results on agreement.

f denotes number of failure-prone processes. n is the total number of processes.

Failure Mode	**Synchronous system (message-passing and shared memory)**	**Asynchronous system (message-passing and shared memory)**
No Failure	agreement attainable; common knowledge attainable	agreement attainable; concurrent common knowledge
Crash Failure	agreement attainable $f < n$ processes	agreement not attainable
Byzantie Failure	agreement attainable $f \leq [(n - 1)/3]$ Byzantine processes	agreement not attainable

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

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.

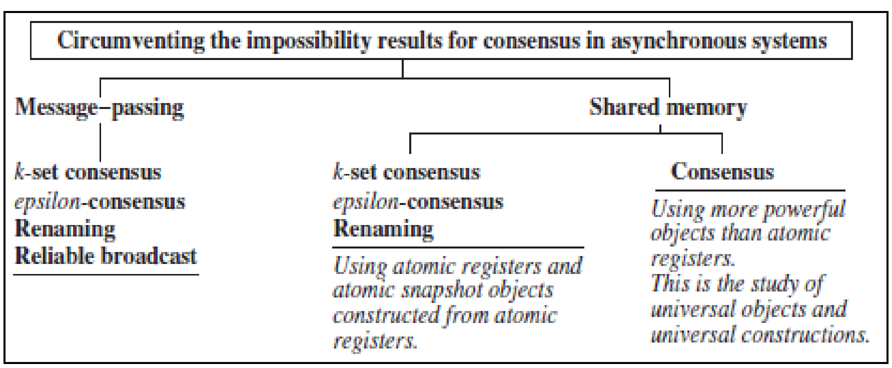
Termination: Each non-faulty process must eventually decide on a value.

Consensus Problem in Asynchronous Systems.

The overhead bounds are for the given algorithms, and not necessarily tight bounds for the problem.

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

Circumventing the impossibility results for consensus in asynchronous systems:



STEPS FOR BYZANTINEGENERALS (ITERATIVE FORMULATION), SYNCHRONOUS, MESSAGE-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) $v_{head(L)}^{tail(L)} \longleftarrow v$; $// \; sizeof(L) + faulty = f + 1$. 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) $v^{L}_{x}(x \neq i, x \notin L) = majority_{y \notin concat(\langle x \rangle, L); y \neq i}(v^{L}_{x}, v_y^{concat(\langle x \rangle, L)})$;

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.

STEPS FOR BYZANTINE GENERALS (RECURSIVE FORMULATION), SYNCHRONOUS, MESSAGE-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.

<u>*Oral_Msg(f)*, where $f > 0$:</u>

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.

<u>*Oral_Msg(0)*:</u>

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.

CODE FOR THE PHASE KING ALGORITHM:

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.

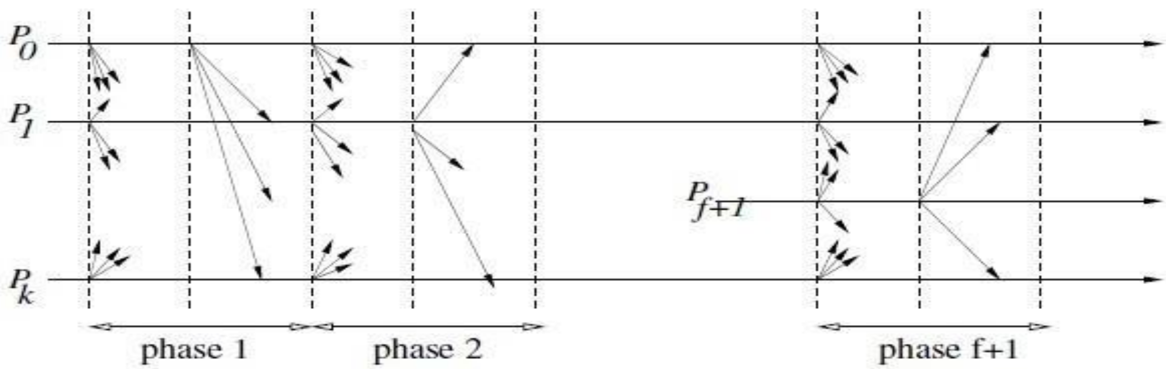


Fig. Message pattern for the phase-king algorithm.

```
(variables)
boolean: v ⟵ initial value;
integer: f ⟵ maximum number of malicious processes, f < ⌈n/4⌉;

(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 ⟵ the value among the v_j that occurs > n/2 times (default if no maj.);
(1f)       mult ⟵ 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 ⟵ majority;
(1m)       else v ⟵ tiebreaker;
(1n)       if phase = f + 1 then
(1o)           output decision value v.
```

PHASE KING ALGORITHM CODE:

(f + 1) phases, (f + 1)[(n - 1)(n + 1)] messages, and can tolerate up to f < dn=4e 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 Pi and Pj will have same estimate of consensus value as Pk does.
- Pi and Pj use their own majority values. Pi 's mult > n=2 + f)
- Pi uses its majority value; Pj uses phase-king's tie-breaker value. (Pi's mult > n=2 + f , Pj 's mult > n=2 for same value)
- Pi and Pj use the phase-king's tie-breaker value. (In the phase in which Pk is non-malicious, it sends same value to Pi and Pj)

In all 3 cases, argue that Pi and Pj end up with same value as estimate

- 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.

CODE FOR THE EPSILON CONSENSUS (MESSAGE-PASSING, ASYNCHRONOUS):

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

Validity: If a non-faulty process Pi decides on some value vi , 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 _ from any other processes estimate.
- # rounds determined by lines (1a)-(1c).

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

(1) Execution at process P_i, 1 ≤ i ≤ n:
(1a) V ⟵ Asynchronous_Exchange(v, 0);
(1b) v ⟵ any element in(reduce^{2f}(V));
(1c) r ⟵ ⌈log_c(diff(V))/ε⌉, where c = c(n − 3f, 2f).
(1d) for round from 1 to r do
(1e)      V ⟵ Asynchronous_Exchange(v, round);
(1f)      v ⟵ new_{2f,f}(V);
(1g) broadcast (⟨v, halt⟩, 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 ⟨x, halt⟩ as value, use x as its input henceforth;
(2d) return the multiset V.
```

TWO-PROCESS WAIT-FREE CONSENSUS USING FIFO QUEUE, COMPARE & SWAP:

Wait-free Shared Memory 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.

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. The crash failure model requires the solutions to be wait-free.

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**$(temp)$.

NONBLOCKING UNIVERSAL ALGORITHM:

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 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, having consensus number k. This trivially follows after the first step.

Any object X with consensus number k is universal in a system with $n \leq k$ processes.

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.

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 an arbitrary consensus object (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.

- There are as many universal objects as there are operations to thread.
- A single pointer/counter cannot be used instead of the array Head. Because reading and updating the pointer cannot be done atomically in a wait-free manner.
- Linearization of the operations given by the sequence number. As algorithm is nonblock