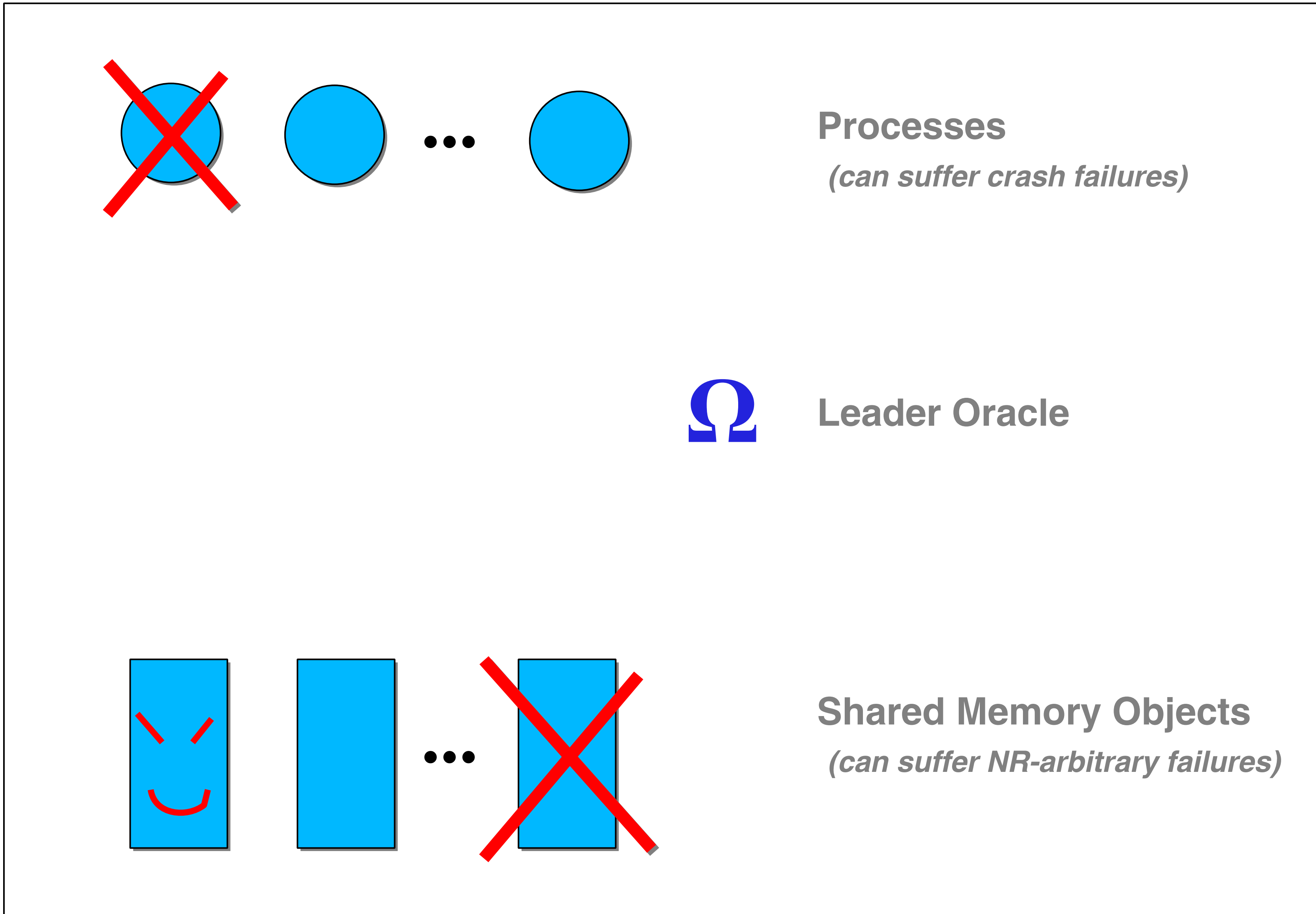
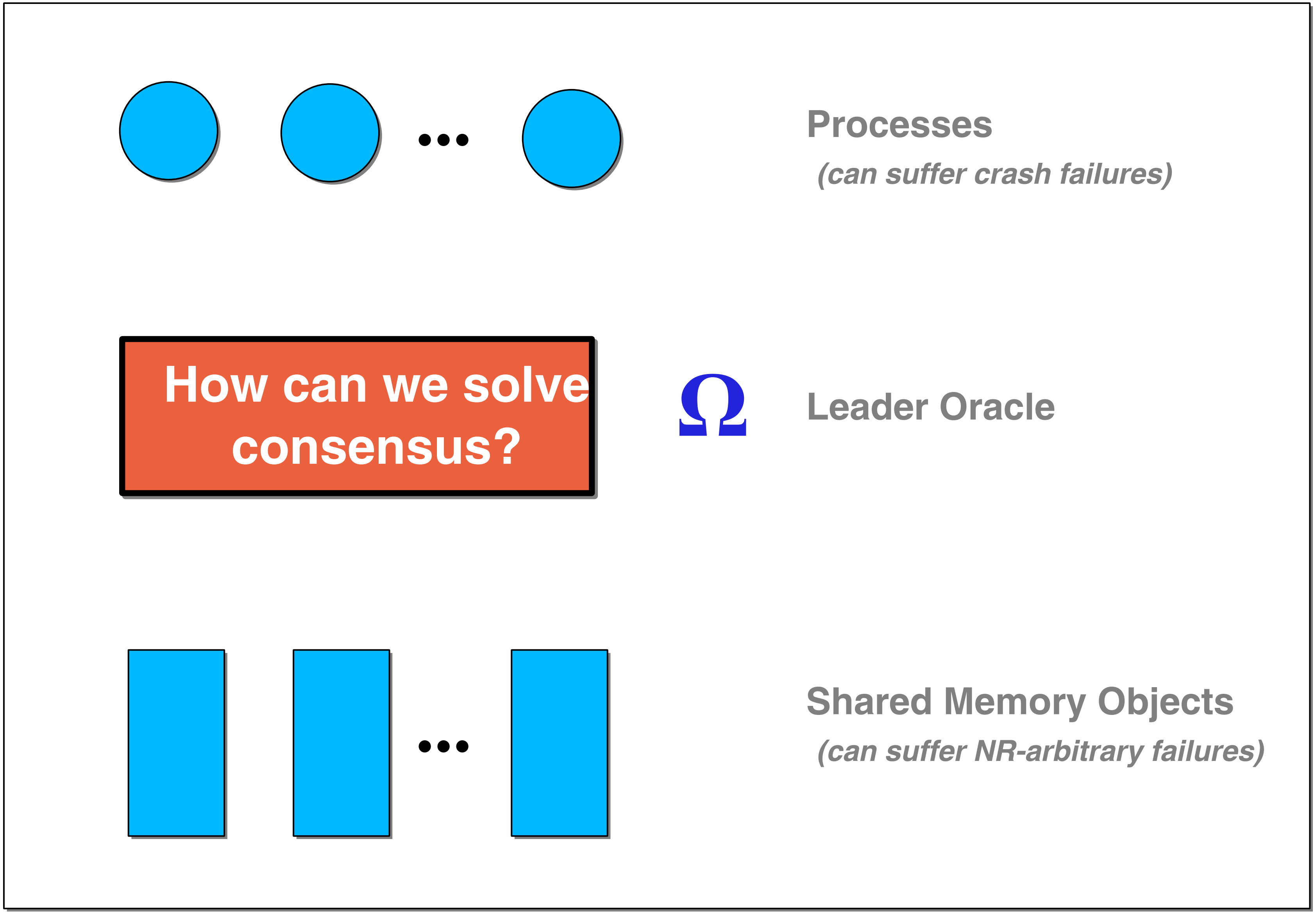
Byzantine Disk Paxos

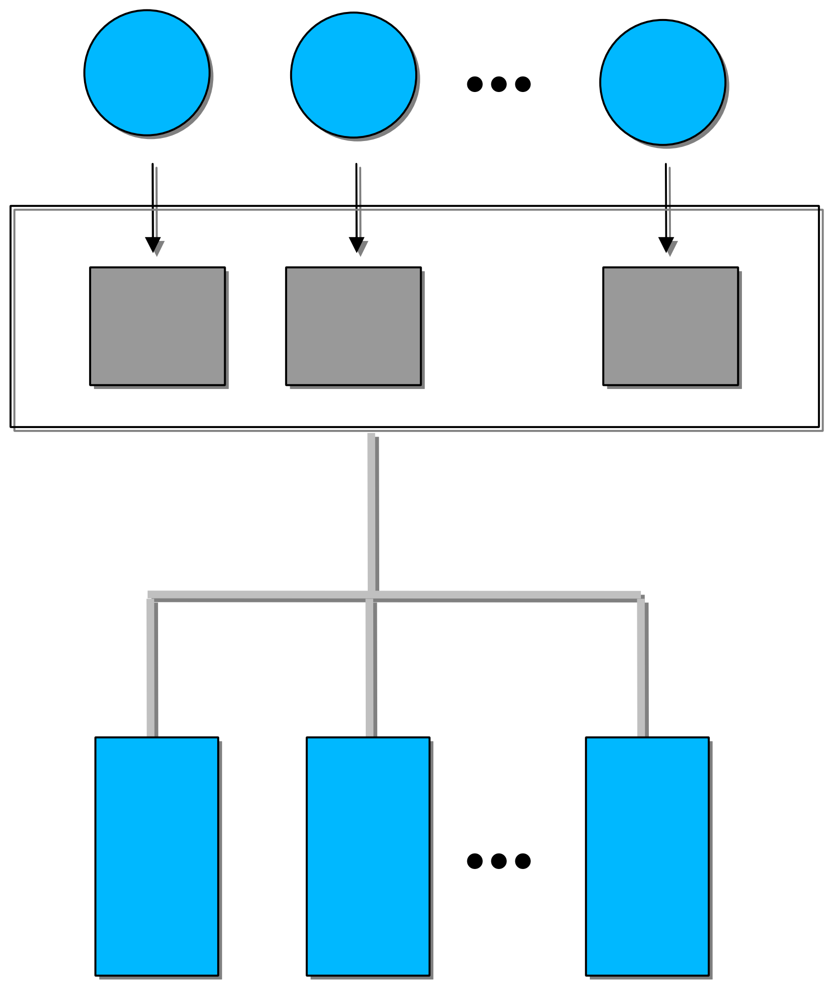
Byzantine Disk Paxos -- *The Setup*



Byzantine Disk Paxos -- *The Setup*



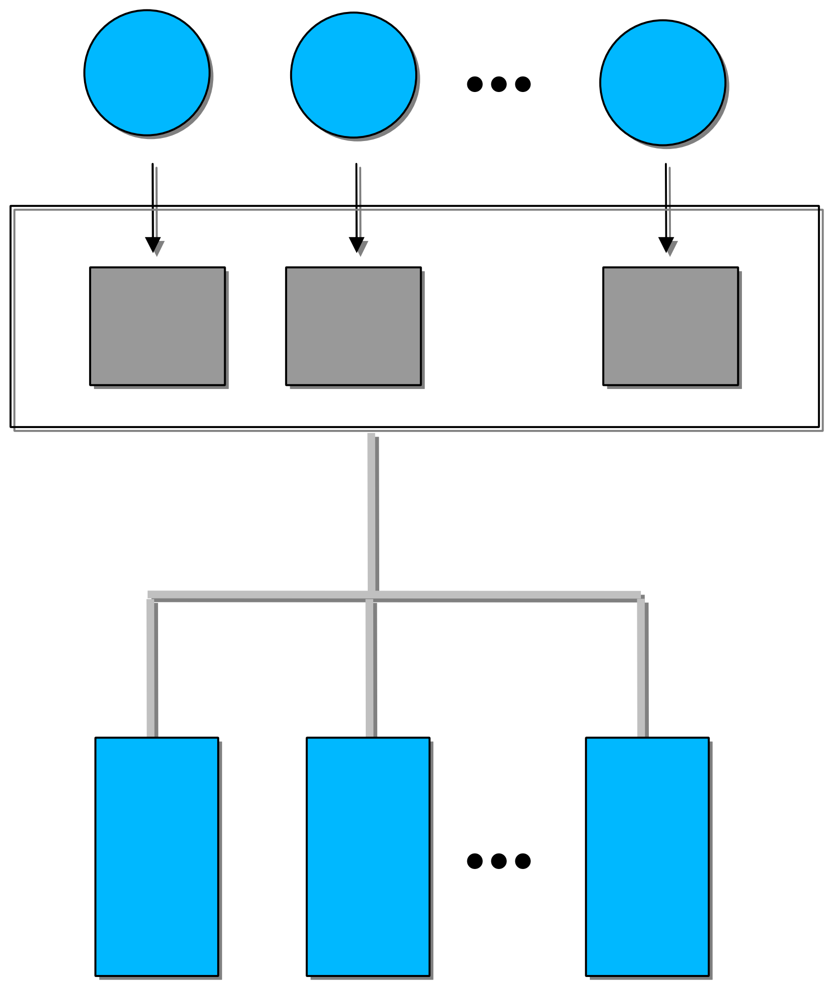
Byzantine Disk Paxos -- *The Basic Idea*



Emulate Reliable Registers:

(We saw this two weeks ago.)

Byzantine Disk Paxos -- *The Basic Idea*



Emulate Reliable Registers:
In ACKM* two types of registers:

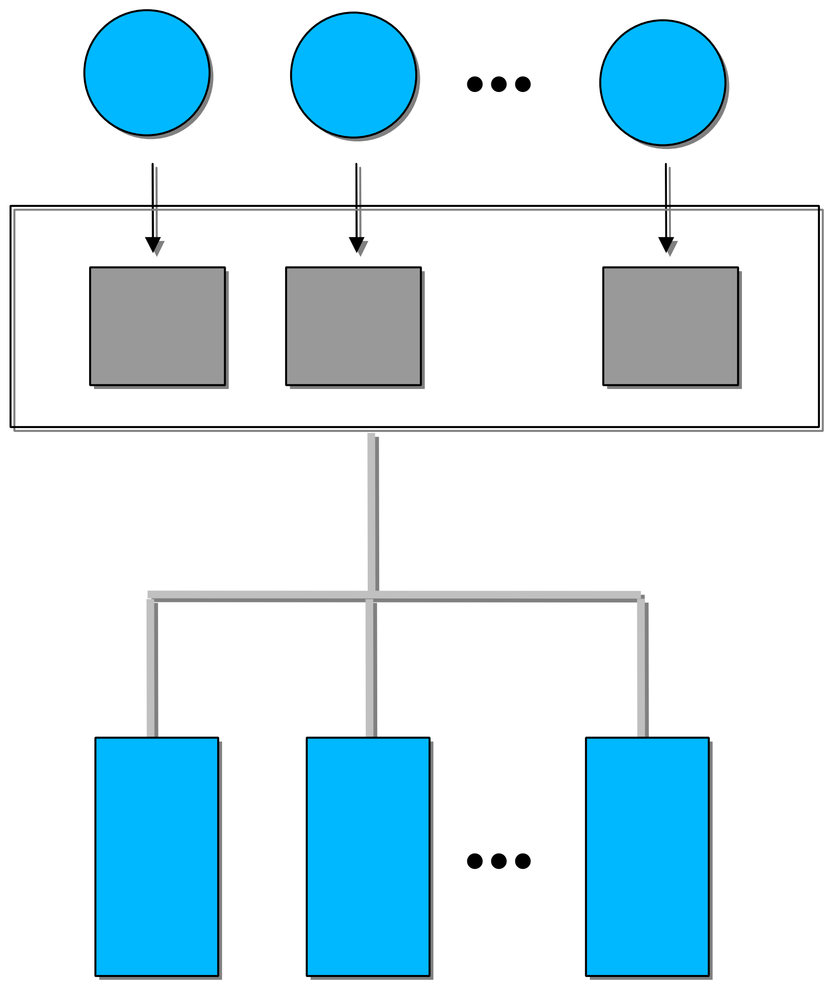
* **Wait-free SWMR safe**

* **FW-termination SWMR regular**

Tolerates **t < n/3** memory object failures – a tight bound.**

* I. Abraham, G. Chockler, I. Keidar, and D. Malkhi. Byzantine Disk Paxos: Optimal Resilience with Byzantine Shared Memory
** J-P. Martine, L. Alvisi, and M. Dahlin. Minimal byzantine storage. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC)*, October 2002.

Byzantine Disk Paxos -- *The Basic Idea*



Emulate Reliable Registers:

In ACKM* two types of registers:

* **Wait-free SWMR safe**

  Can convert to regular using known procedures, and solve with Disk Paxos approach.***

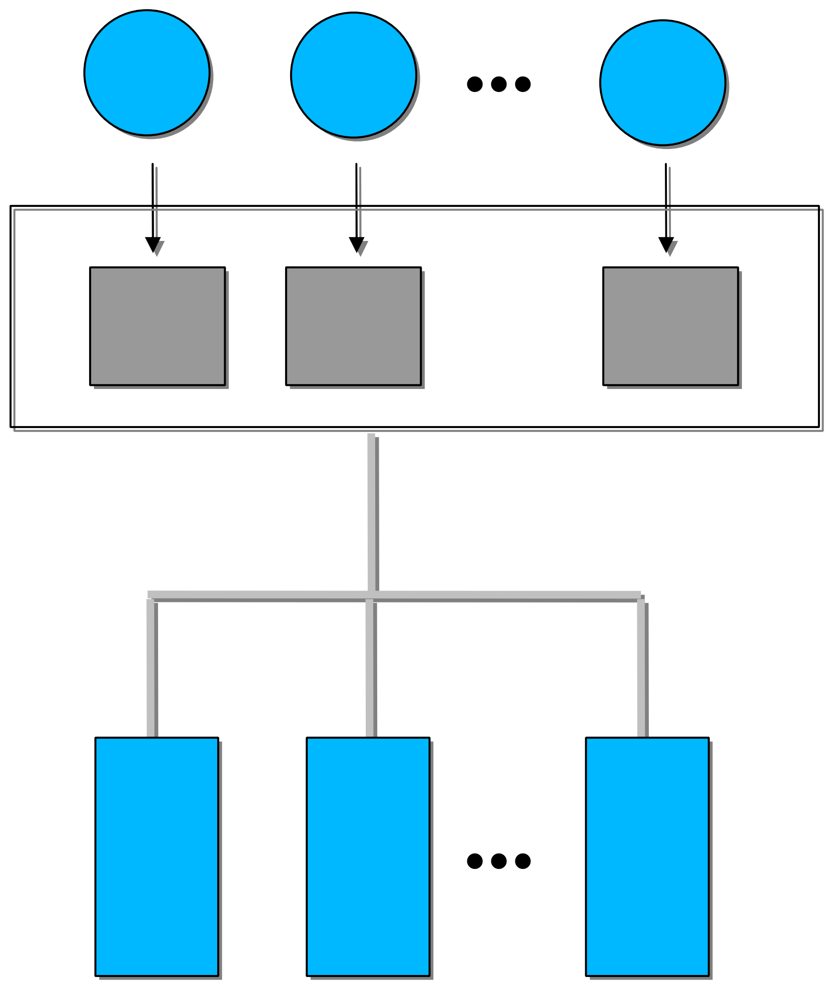
* **FW-termination SWMR regular**

  ACKM provides Disk Paxos style algorithm, and proves it works with FW registers.

* L. Lamport. On interprocess communication – part ii: Algorithms. *Distributed Computing*, 1(2):86-101, 1986

** E. Gafni and L. Lamport. Disk paxos. *Distributed Computing*, 16(1):1-20, 2003.

Byzantine Disk Paxos -- *The Basic Idea*



Emulate Reliable Registers:

In ACKM* two types of registers:

* **Wait-free SWMR safe**
  Can convert to regular using known procedures, and solve with known approach.***

* **FW-termination SWMR regular**
  ACKM provides Disk Paxos style algorithm, and proves it works with FW registers.

* L. Lamport. On interprocess communication – part ii: Algorithms. *Distributed Computing*, 1(2):86-101, 1986
** E. Gafni and L. Lamport. Disk paxos. *Distributed Computing*, 16(1):1-20, 2003.

Based on existing shared memory consensus algorithms:

E. Gafni and L. Lamport. Disk paxos. *Distributed Computing*, 16(1):1-20, 2003

W. K. Lo and V. Hadzilacos. Using failure detectors to solve consensus in asynchronous shared-memory systems. In *Proceedings of the 8th International Workshop on Distributed Algorithms (WDAG)*, pages 280-295. Springer-Verlag, 1994.

Algorithm Setup:

* m processes
* m FW-terminating SWMR regular registers (x1...xm)
* distributed leader oracle

Byzantine Disk Paxos -- *Consensus w/ FW-terminating Registers and Ω*

Pseudo-Pseudo Code for Process i:

```
bal <-- i;                                                        (1)
val <-- <initial value>;                                          (2)

while (true) do                                                   (3)

 if you trust yourself then                                       (4-5)
        reset register by writing <bal,_,_>;                      (6)
        read all registers and store values;                      (7)
        if you have the largest ballot number in read set then    (8)

                choose a proposal value val by examining the read set;   (9-10)
                propose val by writing <bal,val,pc> to register;  (11)
                read all registers (again) and store values;      (12)
                if your proposed ballot number is  largest then   (13)
                        write <bal,val,c> to register;            (14)
                        decide and halt;                          (15)
        increase bal;                                             (16)

 else                                                             (17)

        read register of process you trust;                       (18)
        if value is a decision value then                         (19)
                decide the same and halt;                         (20)
```

Validity:

Obvious, as every proposed value is a process's initial value or a previously proposed value.

Termination:

1) Every fair execution eventually reaches a point after which no more failures occur, and every correct process trusts the same correct process k.

2) After this point, all processes that are not k can do at must two writes (lines 11 and 14) before looping on the non-leader read (line 18).

3) FW-termination then saves the day, as with all processes finishing their writes, k can be guaranteed to finish its reads. It will then continually loop through the leader case, incrementing its ballot number each iteration (line 16) until it decides.

4) Once k decides, no more writes will happen ever again. Therefore, by FW-termination, all the non-k processes will complete their read operations (line 18), see k's decision value, and decide the same.

Agreement (part 1):

1) Assume b1 is the lowest ballot at which some process decides.
   Assume process i decides v1 with this ballot.

2) Process k comes along and proposes v2 with ballot b2 > b1.

3) We will show v2 = v1 which implies agreement as processes
   only decide the value they just proposed with the same ballot.

Use induction on b >= b1:

The base case b = b1 is trivial (unique ballot numbers).

For the inductive step, assume the result holds for b, b1 <= b < b2.

Agreement (part 2):

Back to process k proposing v2 with ballot b2 > b1:

4) Process i decided v1 with ballot number b1. *What does this tell us?* First, process i first proposed v1 with ballot number b1 (line 11), then it read all values (line 12) and its ballot number was still the highest...

5) ...therefore, process k's register clearing write (line 6) did not occur until after process i started its post-proposal read (line 12).

6) This is good because it shows that process k does not do its initial read (line 7) until after process i's proposal *(remember, at this point, process i is at least line 12 on its way to deciding...its proposal occurred at line 11).*

Agreement (part 3):

7) Therefore, process k will read <b1, v1, *> (line 7) so we know:

* Process k's test for existing proposals (line 9) returns true...

* Process k will choose a pre-existing proposal value v' with ballot b' >= b1...

* To get to line 9 the test at line 8 must have been true, so b' <= b2...can reduce to strictly b' < b2.

* To have read <b', v', *> this value must have been proposed at b1 <= b' < b2...

* This brings us back to the induction hypothesis which says v' = v1, so process k will propose value v = v' = v1.