subtle issues can arise. See, for example, the paper *Linearizable Implementations do not Suffice for Randomized Distributed Computation*, by Golab, Higham, and Woelfel, in STOC 2011, pages 373–382.

9.1 Simulations with Different Numbers of Processes

In this section, we consider simulations of one asynchronous shared memory model by another with a different number of processes.

An easy observation is that a wait-free implementation of an object shared by n processes is also a wait-free implementation shared by fewer processes. This leads to the following impossibility result.

Lemma 9.4. *If there is a wait-free linearizable implementation of an object O' with consensus number c' in a model with $n > c$ processes using only registers and instances of objects with consensus number at most c, then $c \geq c'$.*

Proof. Let B be a set of objects with consensus number at most c, let O' be an object with consensus number c', and suppose there is a wait-free linearizable implementation of O' in a model with $n > c$ processes using only registers and objects in B. Because this implementation is wait-free, it also works if there are only $n' = \min\{n, c'\} \leq n$ processes.

Since O' has consensus number c', there is a consensus algorithm for c' processes and, hence, for $n' \leq c'$ processes, using only registers and instances of O'. Simulate this algorithm by replacing each instance of O' with an implementation from registers and instances of objects in B. This simulation solves consensus for n' processes using only registers and objects with consensus number at most c, so $n' \leq c$. This implies $n' < n$ and, thus, $n' = c'$. Hence $c' \leq c$. □

The availability of additional processes is not a problem if the number of processes that can crash does not increase.

Lemma 9.5. *Let M' be a model with $f + 1$ processes and let M be a model with $n > f$ processes and one register, R, in addition to the objects in M'. Furthermore, suppose that, in any execution of M, at most f processes can crash. If there is no solution to consensus in M, then there is no solution to consensus in M'.*

Proof. Suppose there is a (wait-free) solution to consensus in M'. This algorithm can be simulated in M. Specifically, each process is M' is simulated by a different process in M. Just before returning, it writes its output value to R. The initial value of R is not a valid output. The remaining $n - f - 1$ processes in M repeatedly read R until it contains a value different from its initial value.

Since at most f processes crash in any execution, at least one of the processes performing the simulation does not crash. All such processes eventually complete the simulation and write the same output value to R. All of the $n - f - 1$ remaining processes that do not crash will eventually read this value and return it. Thus, there is a solution to consensus in M. □

More generally, an algorithm can be simulated by any number of processes if the number of processes that can crash does not change, using extra registers and test&set objects. The idea is that every simulated process is simulated by every simulating process, rather than by just one simulating process.

Theorem 9.6. *Let M' be a model with $n > f$ processes. Let M be a model with $f + 1$ processes that has n test&set objects and n registers in addition to the objects in M'. If there is no wait-free solution to k-set consensus in M, then there is no solution to k-set consensus in M' that tolerates f process crashes.*

Proof. Suppose there is a solution to k-set consensus in M' that tolerates f process crashes. The algorithm can be simulated in M as follows. Associated with each process q'_j in M', there is one test&set object, T_j, which is initially *unset*, and one register, R_j which will contain a state of the process. Initially R_j contains $\perp$, which is not a valid state. Every object that is in M' has the same initial value in M as it has in M'.

Each process p_i in M tries to choose the input for each process q'_j in M' and simulate the successive steps of its algorithm. It begins by performing test&set on T_j. If it is unsuccessful, it continues on to the next process in M' in round-robin order. If it is successful, it reads R_j. If it contains $\perp$, process p_i writes the initial state of process q'_j into R_j, using its own input as the input to q'_j. Then process p_i resets T_j and continues to the next process in M'. If not, R_j contains the current state of q'_j. If process q'_j is about to return a value, process p_i resets T_j and returns this value. Otherwise, it performs the next step on behalf of q'_j, writes the resulting state of q'_j into R_j, resets T_j, and continues on to the next process in M'.

Each step that is performed on behalf of a process in M' is linearized when it is performed during the execution in M. The test&set object T_j associated with process q'_j is used as a lock to prevent any step of q'_j from being performed more than once. If a process in M crashes when it holds this lock, no further steps of process q'_j are simulated. In other words, process q'_j crashes in the simulated execution. If at most f of the $f + 1$ processes in M crash, then at most f simulated processes crash in the simulated execution. Since the k-set consensus algorithm being simulated tolerates f process crashes, the simulated outputs satisfy k-agreement and validity. Hence, the simulated algorithm solves k-set consensus in M. □

In their paper, *A Completeness Theorem for a Class of Synchronization Objects*, 12th PODC, 1993, pages 159–170, Afek, Weisberger, and Weisman gave a wait-free, linearizable implementation of a test&set object using registers and any objects with consensus number at least two. Combining this fact with Theorem 9.6 for $k = 1$ and Lemma 9.2 implies the following result.

Corollary 9.7. *There is no solution to consensus that tolerates $f \geq 2$ process crashes in a system of $n > f$ processes using only objects with consensus number at most f.*

In particular, there is no algorithm for consensus in a system of $n \geq 3$ processes that tolerates two process crashes using only registers and test&set objects, fetch&add

objects, swap objects, stacks, or queues. There is also no algorithm for consensus in a system of $n \geq 2m-1$ processes that tolerates $2m-2$ process crashes using only registers and m-assignment objects, for $m \geq 2$.

Lemma 9.6 and Corollary 9.7 are from Tushar Deepak Chandra, Vassos Hadzilacos, Prasad Jayanti, and Sam Toueg's paper, *Generalized Irreducibility of Consensus and the Equivalence of t-Resilient and Wait-Free Implementations of Consensus*, SIAM Journal on Computing, volume 34, number 2, 2004, pages 333-357. However, the corollary is false when $f = 1$. In their paper *On the power of shared object types to implement one-resilient Consensus*, Distributed Computing, volume 13, 2000, pages 689-728, Wai-Kau Lo and Vassos Hadzilacos designed a deterministic object, which, together with five registers, can be used by three processes to get a solution to consensus that tolerates one process crash. They also proved that wait-free consensus for two processes cannot be solved using only instances of this object and registers.

9.2 The BG Simulation

The BG simulation uses an approach similar to the proof of Theorem 9.6 for simulating an algorithm by a smaller number of processes, while still tolerating the same number of process crashes. However, it assumes that processes only communicate using single-writer registers.

Instead of using a test&set object as a lock to ensure that each step of each simulated process is performed at most once, it uses an object, called *safe election*, that can be implemented from single-writer registers. It supports two operations, nominate and elect. Each nonfaulty process first performs nominate and then repeatedly performs elect until it gets a result other than $\perp$. The identifier of a process that has completed nominate can also be returned by elect. We will show that, in any execution of safe election, all instances of elect that don't return $\perp$ return the same identifier. We will also show that, if no process crashes while performing nominate, then all processes that do not crash eventually get a result other than $\perp$ from elect.

The implementation of safe election that appears in Figure 9.1 uses one single-writer register $S[i]$ for each process p_i. These are all initialized to 0. In nominate, process p_i writes 1 to $S[i]$ and then reads the other registers. If any of them contains the value 2, it writes 0 to $S[i]$; otherwise, it writes 2. In elect, a process reads the registers in order until it sees a nonzero value. If it reads the value 1, it returns $\perp$. If it reads 2 from register $S[\ell]$, it returns the value ℓ, indicating that process p_ℓ has been elected.

From the code, it follows that no process completes nominate until some register contains the value 2. Furthermore, once a register contains the value 2, it never changes value. Thus, if a process does not crash while performing elect, one of the tests will be successful and it will return with $\perp$ or with a process id. A process has 1 in its register only when it is performing nominate. Therefore, if no process crashes while performing nominate, then, eventually, no register contains 1. Any instance of elect that starts after this point will not return $\perp$.

```
nominate
S[i] ← 1
for j ∈ {0, ..., n − 1} − {i} do
    if S[j] = 2
    then S[i] ← 0
         return
S[i] ← 2
return

elect
for ℓ ← 0 to n − 1 do
    s ← S[ℓ]
    if s = 1 then return(⊥)
    if s = 2 then return(ℓ)
```

Figure 9.1. Safe election, code for process p_i

Consider any execution of safe election in which each process begins by performing nominate and thereafter only performs elect. Let p_j be the process with smallest index that writes 2 to its register. Then no process returns any value smaller than j from elect. Let C be the first configuration in which some register contains 2. Any process that starts nominate after configuration C will read 2 from some register and, hence, will not write 2 to its own register. Therefore, in configuration C, register $S[j]$ contains 1 or 2. Since no process starts elect until after C, every call of elect returns either $\perp$ or j. Thus, all processes that return from elect with a value other than $\perp$ return the same value.

Let M' be an asynchronous shared memory model in which n processes communicate using single-writer registers. It suffices to assume each process q'_j in M' has exactly one single-writer register R'_j to which it can WRITE, since multiple single-writer registers with the same writer can be combined into one using different fields or using tags.

Suppose there is a solution to k-set consensus in M' that tolerates f process crashes, where $k \leq f < n$. We will show how to obtain a wait-free simulation of this algorithm in an asynchronous shared memory model M with $f + 1$ processes that communicate using single-writer registers.

As in the simulation in the proof of Theorem 9.6, each process p_i in M tries to choose the input for each process q'_j in M' and simulate the successive steps of its algorithm. A separate set of n single-writer registers, $\{S[j, t, i] \mid i = 0, \ldots, n-1\}$, is used to perform safe election for the t'th simulated step of process q'_j.

Unlike test&set, safe election is not an atomic operation. If a process p_i crashes while it is performing multiple instances of nominate, it could cause the simulations of those processes to block. To prevent this from happening, once process p_i begins to perform nominate, it completes it before doing anything else. Since the worst-case step complexity of nominate is $n + 1$, this cannot prevent p_i from simulating other processes in M'.

A process p_i might continually perform elect with result $\perp$ in some instance of safe election, because some other process has crashed or will eventually crash. To ensure it continues to make progress, process p_i might have to perform many instances of safe election concurrently. However, if process p_i crashes after being elected to perform the next steps of many simulated processes, but before it performs them, the simulations of all those processes will be blocked. Instead, to simulate the t'th step of process q'_j, each process p_i performs this step and records the resulting state of q'_j (including the contents of R'_j) in a single-writer register $R[j, t, i]$ before performing the t'th instance of safe election for process q'_j. Then, even if the elected process crashes while it is performing elect, the remaining processes in M can continue with the simulation of process q'_j.

Suppose that, during an execution of the instance of safe agreement for the t'th step of process q'_j, a process is elected that changes the simulated value of R'_j from v to v'. Later, it is possible that another (slow) process crashes while performing nominate in this same instance of safe agreement. Then any process that performs elect afterwards will return $\perp$ and, hence, is blocked from finding out which process was elected. This can make it impossible to linearize the simulated execution, since one process could simulate reading v' from R'_j and, later, another process could simulate reading v from R'_j. To prevent this from happening, there is an n-component single-writer snapshot object A_j for each process q'_j in M'. (Note that there is a wait-free, linearizable implementation of a single-writer snapshot object from single-writer registers.) We may assume that, initially, each component of A_j contains $\perp$. Immediately after a process p_i returns with id ℓ from elect in the t'th instance of safe election for process q'_j, it updates component i of the snapshot object A_j with the value (t, ℓ). A simulating process can then scan A_j to determine the last step of process q'_j that has been simulated and which simulating process was elected for that step.

A process p_i simulating a process q'_j first does a scan of A_j. If it sees that A_j has never been updated, it writes the initial state of process q'_j to register $R[j, 0, i]$ (including the initial value of R'_j) using its own input as the input to process q'_j. Then it performs nominate and one call to elect in the instance of safe election for the initialization of process q'_j using the set of registers $\{S[j, 0, i] \mid i = 0, \ldots, n-1\}$. If $\perp$ is returned, it temporarily stops simulating q'_j and starts or continues simulating the next process in M', in round robin order. If p_i ever returns from elect with a value $\ell \neq \perp$ in this instance of safe election, it then updates component i of the snapshot object A_j with the pair $(0, \ell)$ to finish its simulation of the initialization of process q'_j.

If process p_i sees that A_j has been updated at least once, it finds the pair (t, ℓ) contained in its components with the largest value of t. Then it reads the state of process q'_j after its t'th step, as recorded in $R[j, t, \ell]$ by the process p_ℓ elected for this step. If the next step of process q'_j is to return value v, then process p_i returns value v and performs no further steps in the simulation. If the next step of process q'_j is a WRITE, process p_i determines the resulting state of process q'_j and writes this state (including the value written by process q'_j) to register $R[j, t+1, i]$. If the next step of process q'_j is a READ of the single-writer register R'_r of process q'_r, process p_i scans the snapshot object A_r.

From this, p_i determines t_h, the number of steps of process q'_r that have been simulated, and which process, p_h, was elected in the instance of safe agreement for the last of these steps. Then process p_i reads $R[r, t_h, h]$ and, from it, determines the simulated contents of R'_r at the end of step t_h. Finally, process p_i determines the resulting state of process q'_j and writes this state to $R[j, t+1, i]$. For both READS and WRITES, process p_i next performs nominate and one call to elect in the instance of safe election for step $k+1$ by process q'_j, using the set of registers $\{S[j, t+1, i] \mid i = 0, \ldots, n-1\}$. If $\perp$ is returned, it temporarily stops simulating q'_j and starts or continues simulating the next process in M', in round robin order. If p_i ever returns from elect with a value $\ell \neq \perp$ in this instance of safe election, it then updates component i of the snapshot object A_j with the pair $(k+1, \ell)$ as its final piece of the simulation of step $k+1$ by process q'_j.

After finishing the simulation of the initialization of process q'_j or a step by process q'_j, process p_i scans A_j and starts simulating another step of process q'_j.

Each READ that is performed on behalf of a process q'_j in M' is linearized when the process elected for that step does its scan of A_j. If step t of a simulated process is a WRITE, it is linearized the first time any process updates the snapshot object A_j with a pair (t, ℓ). Note that all processes that finish this instance of safe election return from elect with the same value, namely the index of the process elected for this step.

If a process in M crashes while it is performing nominate as part of its simulation of step t by process q'_j, it can only block further steps of process q'_j from being simulated. Thus, if at least one process (in M) does not crash, then at most f simulated processes crash in the simulated execution. Since the k-set consensus algorithm being simulated tolerates f process crashes, the simulated outputs satisfy k-agreement and validity. Hence, the simulated algorithm solves k-set consensus in M.

Thus, we have shown the following result.

Theorem 9.8. *Let M and M' be asynchronous shared memory models in which processes communicate using single-writer registers. Suppose M has $f+1$ processes and M' has $n > f \geq k$ processes. If there is no wait-free solution to k-set consensus in M, then there is no solution to k-set consensus in M' that tolerates f process crashes.*

Corollary 9.9. *There is no solution to k-set consensus for $n > f \geq k$ processes that tolerates f process crashes in an asynchronous shared memory model in which processes communicate using single-writer registers.*

Proof. Suppose there is a solution to k-set consensus in M' that tolerates f process crashes. Since any algorithm for k-set consensus is also an algorithm for f-set consensus, it follows from Theorem 9.8 that there is a wait-free algorithm for $f+1$ processes that solves f-set consensus using only single-writer registers. This contradicts Theorem 2.10. □

The BG simulation is due to Borowsky and Gafni and originally appeared in their paper *Generalized FLP impossibility result for t-resilient asynchronous compuations*, Proceedings of the 25th ACM Symposium on Theory of Computing, 1993, pages 91–100. Its proof of correctness appears in *The BG Distributed Simulation Algorithm*, by Borowsky, Gafni, Lynch, and Rajsbaum, Distributed Computing, volume 14, 2001, pages 127–146.

9.3 Round by Round Simulations

Round-by-round simulations are used to derive a lower bound on the number of rounds to solve a problem in a synchronous message-passing model M from the impossibility of that problem in an asynchronous shared-memory model M' with the same number of processes, in which processes communicate using registers. Specifically, they show how to simulate any f-round execution in model M, in which at most one process crashes each round, using model M', in which at most one process crashes.

We consider a synchronous broadcast model M. In each round, a process sends the same message to all processes, receives messages from all processes that have not crashed, and possibly receives some messages from the processes that crashed during the round.

This model can simulate a model in which a process can send different messages to different processes in the same round by broadcasting the concatenation of all the messages and their intended recipients in the round. This does not change the number of rounds of communication.

The round by round simulation will use a simple approximate agreement object. This object supports one operation, propose(v), where $v \in \{0, 1\}$, which returns a value in $\{0, \frac{1}{2}, 1\}$ such that, in every execution, the return values satisfy validity and $\frac{1}{2}$-agreement. Thus, if all inputs to propose are the same, then this is the only value that is returned. Furthermore, there is no execution in which 0 and 1 are both returned.

The implementation in Figure 9.2 uses two registers, R_0 and R_1, both initially 0. If the input value of a process is 0, it writes 1 to R_0; otherwise it writes 1 to R_1. Then it reads the other register. If the other register is still 0, then the process outputs its input value; otherwise, it outputs $\frac{1}{2}$. If all processes have the same input value v, they

```
propose(x)
R_x ← write 1
if R_{1−x} = 0 then return x
               else return 1/2
```

Figure 9.2. Approximate agreement object, code for process p_i

only write to R_v and, hence, they can only read 0 from R_{1-v}. Thus, the only value they output is v. Otherwise, some process has input 0 and some other process has input 1. This ensures validity, since the only possible output values are 0, 1, and $\frac{1}{2}$.

Suppose register R_v is the first register that is written to in some execution. Then every process with input value $1 - v$ will read 1 from R_v and will output $\frac{1}{2}$. Hence, only values in $\{v, \frac{1}{2}\}$ are output and they are within $\frac{1}{2}$ of one another. Thus, $\frac{1}{2}$-agreement is ensured.

Theorem 9.10. *Any n-process consensus algorithm in model M that tolerates $f < n$ process crashes requires more than f rounds.*

Proof. Suppose there is an algorithm for n-process consensus in model M that tolerates $f < n$ process crashes and requires at most f rounds. We will simulate an execution of this algorithm in model M' to obtain a contradiction.

Each process q_i of M' simulates a different process p_i of M, using its own input as the input to p_i. The processes simulate the rounds of the algorithm in order. Since one process in M' can crash, processes must finish the simulation of a round when at least $n-1$ processes have participated in it.

The idea is that if some process q_i is slow to participate in round r or it crashes before announcing the message p_i broadcast in round r, other processes will propose that process p_i is faulty. If the processes agree that p_i is faulty, they will stop simulating it. However, it is not necessary that the simulating processes agree in the same round. It suffices that some processes think agreement is reached in one round and the remainder think it is reached in the subsequent round. This simulates the situation in which process p_i fails during the round after sending messages to the second set of processes.

Each process, q_i locally maintains a set $faulty_i$ of processes in M that it proposes to be faulty. Initially, this set is empty. Once a process is added to this set, it is never removed.

For each round, there is a shared array of single-writer registers, $M_r[0 \ldots n-1]$ all of whose entries are initially empty and a shared array of approximate agreement registers in their initial state $A_r[0 \ldots n-1]$.

At the beginning of the simulation of round $r \geq 1$, process q_i writes into $M_r[i]$ the message that process p_i broadcasts in round r, or a special value indicating that q_i agrees process p_i is faulty. Then process q_i then repeatedly performs collect on M_r until it has seen a view with n nonempty components or it has twice seen a view with $n-1$ nonempty components. If q_i does not fail, it will happen eventually, since at most one process fails. This gives a snapshot of M_r. Note, every process that sees $n-1$ nonempty components in its last collect of M_r sees the same empty component.

For all $j \in \{0, \ldots, n-1\}$, if $j \in faulty_i$ or process q_i sees that $M_r[j]$ is empty, then process q_i performs propose(0) on $A_r[j]$; otherwise it performs propose(1) on $A_r[j]$. If it returns 0 to process q_i, then q_i agrees that process p_j has crashed, adds j to $faulty_i$, and does not simulate the receipt of a message from process p_j to process p_i for $j \neq i$. If it returns1 or $\frac{1}{2}$ to process q_i, then at least one process saw that $M_r[j]$ was nonempty, so process q_i can read $M_r[j]$ to simulate the receipt of a message from process p_j to process p_i. In the latter case, q_i adds j to $faulty_i$ and will henceforth propose that process p_j is faulty.

Note that if $A_r[j]$ returns 0 to any process, then it returns either 0 or $\frac{1}{2}$ to every process, so every process q_i will add j to $faulty_i$. Hence, in all subsequent rounds r', no process will perform propose(1) on $A_{r'}[j]$ and, by validity, $A_{r'}[j]$ will return 0 to every process. Thus, each process will agree that process p_i is faulty in either round r or round $r+1$. □

9.4 Undecidability of Consensus Number

In sequential computation, a common way to prove that a problem is undecidable is to give a reduction to it from a problem already known to be undecidable, such as the halting problem. We can use the same technique for showing the unsolvability of distributed computing problems. Here is one example.

Theorem 9.11. *There is no algorithm that, given the sequential specifications of an object, decides whether its consensus number is 1.*

Proof. To obtain a contradiction, suppose there is such an algorithm. We will use it to solve the halting problem for one-tape Turing machines with initially blank tape.

Given a deterministic Turing machine M, let $\mathcal{C}$ be the set of all configurations of M and let C_0 be the configuration when M is in its initial state and its tape is blank. Define the object $\mathcal{O}$ whose value set is $\mathcal{C} \times \{true, false\}$, whose intial value is $(C_0, false)$, and which supports one operation, next. This operation takes no input and returns a value in $\{0, 1, 2\}$. When $\mathcal{O}$ has value $(C, flag)$, next behaves as follows:

- If $flag = false$ and C is not a final configuration of M, then next($\mathcal{O}$) returns 0 and the new value of $\mathcal{O}$ is $(C', false)$, where C' is the configuration that results when M takes one step starting from configuration C.
- If $flag = false$ and C is a final configuration of M, then next($\mathcal{O}$) returns 1 and the new value of $\mathcal{O}$ is $(C, true)$.
- If $flag = true$ next($\mathcal{O}$) returns 2 and the value of $\mathcal{O}$ remains unchanged.

Suppose that M halts starting from configuration C_0. Then the algorithm in Figure 9.3 solves wait-free consensus for two processes using object $\mathcal{O}$ and two single-writer registers, R_0 and R_1.

```
propose(x_i)
R_i ← x_i
repeat u ← next(O) until u ≠ 0
if u = 1 then return x_i
         else return R_{1−i}
```

Figure 9.3. Two-process wait-free consensus, code for process p_i

The value decided in any execution of this algorithm is the input value of the process that first performs next($\mathcal{O}$) after $\mathcal{O}$'s first component is a final configuration. In this case, the consensus number of $\mathcal{O}$ is at least 2.

Now suppose that M does not halt starting from configuration C_0. Then next($\mathcal{O}$) always returns 0. If there was an algorithm that solved wait-free consensus for two processes using only registers and copies of object $\mathcal{O}$, then there would also be an algorithm

that solved wait-free consensus for two processes using only registers: simply replace each occurrence of next by the constant 0. But this is impossible. Therefore, the consensus number of $\mathcal{O}$ is 1.

Therefore M halts starting from configuration C_0 if and only if the consensus number of $\mathcal{O}$ is not 1. Since the halting problem is undecidable, it follow that deciding whether $\mathcal{O}$ has consensus number 1 is also undecidable. □

It follows that there is no algorithm to compute the consensus number of an object. However, for certain classes of objects, for example, deterministic objects with finite value sets that support only read-modify-write primitives, there is an algorithm that decides whether a given object has consensus number at least n. This result is by Eric Ruppert, *Determining Consensus Numbers*, SIAM J. Comput., volume 30, number 4, 2000, pages 1156–1168. The proof that determining the consensus number of objects is, in general, unsolvable is from *Some Results on the Impossibility, Universality, and Decidability of Consensus* by Prasad Jayanti and Sam Toueg, which appeared in WDAG 1992, pages 69–84.