Using Failure Detectors to Solve Consensus in Asynchronous Shared-Memory Systems

(Extended Abstract)

Wai-Kau Lo* and Vassos Hadzilacos**

Department of Computer Science
University of Toronto
6 King's College Road
Toronto, Ontario
Canada M5S 1A4

Abstract. Chandra and Toueg proposed a new approach to overcome the impossibility of reaching consensus in asynchronous message-passing systems subject to crash failures [6]. They augment the asynchronous message-passing system with a (possibly unreliable) *failure detector*. Informally, a failure detector provides some information about the processes that have crashed during an execution of the system. In this paper, we present several Consensus algorithms using different types failure detectors in asynchronous *shared-memory* systems. We also prove several lower bounds and impossibility results regarding solving Consensus using failure detectors in asynchronous shared-memory systems.

1 Background and Overview of Results

It is well-known that there is no deterministic algorithm for Consensus in asynchronous distributed systems, even if a single process may crash; this result applies both to message-passing and shared-memory systems [9, 7, 11].[3] This impossibility result has motivated the use of randomisation to solve Consensus. Many randomised Consensus algorithms have been devised for both message-passing and shared-memory asynchronous systems [4, 3].

Recently, Chandra and Toueg [6] introduced another approach to overcome the impossibility of reaching Consensus in asynchronous message-passing systems subject to crash failures. In their approach, the asynchronous model is augmented with a (possibly unreliable) *failure detector*. Informally, a failure detector, denoted $\mathcal{D}$, consists of a collection of modules, one associated with each process. The failure detector module associated with process p is denoted $\mathcal{D}_p$, and can be accessed only by p. At any time, p can query its local failure detector

* Supported by a Canadian Commonwealth Scholarship.

** Supported, in part, by a grant from the Natural Sciences and Enginerring Research Council of Canada.

[3] Unless stated otherwise, throughout this paper we assume that shared memory consists only of atomic read-write registers.

module $\mathcal{D}_p$, and $\mathcal{D}_p$ will return to p a set of processes that it suspects to have crashed at that moment. A failure detector $\mathcal{D}$, however, may make "mistakes"; e.g., $\mathcal{D}_p$ may suspect a process q that has not crashed. If $\mathcal{D}_p$ learns, at a later time, that process q is still alive, it can remove q from its set of suspects.

Chandra and Toueg classify failure detectors according to their *completeness* and *accuracy* properties. Loosely speaking, these specify, respectively, lower and upper bounds on the set of processes that a failure detector can suspect. Two completeness properties are defined:

– Strong (Weak) Completeness: There is a time after which every process that crashes is permanently suspected by *every* (*some*) correct process.

Similarly, we have Strong and Weak Accuracy, defined as follows:

– Strong (Weak) Accuracy: *Every* (*some*) correct process is never suspected.

In addition, for each of the two accuracy properties, there is an eventual version, which states that there is a time after which the corresponding accuracy property holds. For instance, Eventual Weak Accuracy states that there is a time after which some correct process is never suspected.

A failure detector can be specified by choosing a completeness and an accuracy property. For example, an *Eventual Weak* failure detector, denoted $\diamond\mathcal{W}$, satisfies Weak Completeness and Eventual Weak Accuracy. Note that the quality of information about failures that such a failure detector provides can be quite poor: A correct process can be permanently suspected by all other processes; it can also be suspected and then not suspected repeatedly (infinitely often) by the other processes.

Chandra and Toueg [6] define a host of different failure detectors by considering different combinations of completeness and accuracy properties. For each failure detector they consider they give an algorithm for Consensus that uses that failure detector.

In a related paper, Chandra, Hadzilacos and Toueg [5], prove that $\diamond\mathcal{W}$ is, in fact, the weakest failure detector that can be used to solve Consensus. More precisely, they show that *any* failure detector $\mathcal{D}$ that can be used to solve Consensus can also be used to emulate $\diamond\mathcal{W}$. Thus, $\mathcal{D}$ must provide at least as much information about failures as does $\diamond\mathcal{W}$, and $\diamond\mathcal{W}$ is therefore the weakest failure detector that can solve Consensus.

All of the work on failure detectors mentioned above is in the context of asynchronous *message-passing* systems. Given the strong analogies in the results regarding Consensus between message-passing and shared-memory systems (Consensus is not solvable in either by means of deterministic algorithms, but solvable in both by means of randomised algorithms), it is natural to ask whether, and to what extent, the analogies persist when we introduce failure detectors. This is the subject of this paper. We find that many of the results cited above extend to shared-memory systems, although some discrepancies exist. Thus, our results highlight both the similarities and the differences between shared-memory and message-passing distributed systems, and are part of the effort to understand how these two fundamental models of distributed computation relate to one another.

In this paper we assume that only processes may fail, and do so by crashing, i.e., halting prematurely. In particular, the shared registers behave correctly. Problems related to tolerating faulty registers are discussed in [2, 10]. We now give an overview of our results. First, some definitions (cf. [6]): A *Strong* failure detector satisfies the Strong Completeness and Weak Accuracy properties; an *Eventual Strong* failure detector satisfies Strong Completeness and Eventual Weak Accuracy. Similarly, a *Weak* failure detector satisfies Weak Completeness and Weak Accuracy; an *Eventual Weak* failure detector satisfies Weak Completeness and Eventual Weak Accuracy.

In this paper we present a Consensus algorithm that uses a Strong failure detector, and one that uses an Eventual Strong failure detector. Both algorithms use n 1-writer-n-reader registers; in the first the registers are of bounded size, while in the second they are unbounded. Both algorithms are wait-free; i.e., they can tolerate up to $n-1$ faulty processes. It is noteworthy that in *message-passing* systems any algorithm that solves Consensus using an Eventual Strong failure detector requires that a majority of the processes be correct [6]. Thus, our Consensus algorithm for the Eventual Strong failure detector reveals an inherent difference between message-passing and shared-memory systems. Intuitively, this is a reflection of the fact that in an asynchronous message-passing system, a message may be delayed for arbitrarily long; thus whether the process sending the message has taken the corresponding step can remain hidden from the rest of the processes for an unbounded amount of time. In contrast, in shared-memory systems, when a process writes a value into a register, the fact that this has occurred cannot be hidden from any process that subsequently reads that register. The significance of this behaviour was originally noted in [7], in the closely related context of *partially synchronous* message-passing systems.

Chandra and Toueg showed how to emulate a Strong (resp. Eventual Strong) failure detector using a Weak (resp. Eventual Weak) one in message-passing systems [6]. The algorithm that accomplishes these emulations can be adapted to shared-memory systems in a straightforward manner, and we therefore get wait-free Consensus algorithms that use these two weaker failure detectors.

In addition to the algorithms, we prove some lower bounds and impossibility results about reaching Consensus using a Strong failure detector. First, we show that if only 1-writer-n-reader registers are allowed, then any wait-free Consensus algorithm for n processes that uses a Strong failure detector must use at least n registers. Thus, as far as the number of 1-writer-n-reader registers is concerned, our algorithms are optimal.

Our Consensus algorithms require the shared registers to be initialised to specific values. It is interesting to ask whether it is possible to devise algorithms that can solve Consensus *without* requiring such initialisation. We prove that this is impossible if the correct processes are required to eventually *halt* after deciding. (Our Consensus algorithms satisfy this additional requirement.) On the other hand, if the correct processes are allowed to remain active forever after deciding, then it is possible to solve Consensus using a Strong failure detector without requiring that shared memory be initialised.

Finally, we show that the Eventual Weak failure detector is the weakest one that can solve Consensus in asynchronous shared-memory systems, just as it is in message-passing systems. The proof technique follows closely the one presented in [5], except for some technical difficulties that arise as a result of the difference between message-passing and shared-memory asynchronous systems that was pointed out previously.

The rest of the extended abstract is organised as follows: In Section 2, we briefly describe our model of computation. In Section 3, we present our Consensus algorithms that use different types of failure detectors. Section 4 contains some lower bounds and impossibility results. We conclude in Section 5 with a discussion of some open problems.

2 Model of Computation

Our model of computation follows [5] and [11]. In this extended abstract we only sketch the main features of the model, without going into all the formal details.

An asynchronous shared-memory system with a failure detector consists of: (1) a set of shared atomic read/write registers R; (2) a set of n asynchronous processes P; and (3) a failure detector $\mathcal{D}$. To simplify the presentation, we assume the existence of a discrete global clock with the set of natural numbers, denoted $\mathcal{T}$, as the domain of its clock ticks. Processes do not have access to this clock.

2.1 Failure Detectors

A *failure pattern* F is a function from $\mathcal{T}$ to 2^P, where $F(t)$ denotes the set of processes that have crashed by time t; we require that for all $t \in \mathcal{T}$, $F(t) \subseteq F(t+1)$. We say that p *crashes* in F if $p \in F(t)$, for some $t \in \mathcal{T}$; otherwise we say that p is *correct* in F.

Let $\mathcal{R}$ be a set of *failure detector values.* These represent the values that failure detector modules can return.[4] A *failure detector history* H is a function from $P \times \mathcal{T}$ to $\mathcal{R}$; intuitively, $H(p,t)$ is the value of p's local failure detector module at time t in a particular execution. A *failure detector* $\mathcal{D}$ is a function that maps each failure pattern F to a set of failure detector histories. Informally, if $H \in \mathcal{D}(F)$ then H is one of the possible behaviours of the failure detector $\mathcal{D}$ when the failure pattern is F. Each completeness and accuracy property described in Section 1 implicitly defines a set of allowable failure detector histories that correspond to a given failure pattern. Thus, we can specify a failure detector (in the formal sense just stated) by fixing a completeness and an accuracy property.

[4] Typically the failure detector returns a set of "suspected" processes, in which case $\mathcal{R} = 2^P$. The greater generality allows us to model failure detectors that return values like "p and either q or q' has crashed".

2.2 Algorithms Using Failure Detectors

An *algorithm* **A** for n processes using a failure detector $\mathcal{D}$ (in an asynchronous shared-memory system) is modeled as n (possibly infinite state) deterministic automata, one for each of the processes. Let $\mathbf{A}_p$ be the automaton running on process p. A *configuration* C of the system (w.r.t. algorithm **A**) consists of the states of each automaton and each register.

Informally, in a step of algorithm **A**, some process p atomically executes one of the following operations: (1) read a value from a register, (2) write a value into a register, or (3) query the local failure detector module. The operation that a process executes next, and the value that it writes, in the case of a write operation, depends only on its local state. The new state of the process after it executes a step is determined by its present state and, in the case of a read or query operation, by the value of the register read or the value returned by the failure detector.

We represent a step of process p by the triple (p, o, v); o can be one of $read(r)$, $write(r)$ (where r is a register) or $query$, and represents the operation that p executed in the step; and v is the value read or written (if $o = read(r)$ or $write(r)$), or the value returned by the failure detector (if $o = query$). We say that the step (p, o, v) is *applicable* to configuration C if the following holds: Let s be the state of p in C. The automaton $\mathbf{A}_p$ defines o to be the next operation of p when in state s; furthermore, if o is $read(r)$, then v is the value of r in C, and if o is $write(r)$ then v is the value which $\mathbf{A}_p$ specifies that p should write into register r when in state s. If step e is applicable to C we denote by $e(C)$ the configuration that results after we apply e to C. Configuration $e(C)$ differs from C only in the state of the process that takes step e, and in the value of the register written by that process, if e is a write step.

A *schedule* S is a finite or infinite sequence of steps of the algorithm. $S[i]$ denotes the ith step of S. A schedule S is applicable to a configuration C if and only if S is the empty sequence, or $S[1]$ is applicable to C, and $S[2]$ is applicable to $S[1](C)$, etc.

A *run* of algorithm **A** in a system using failure detector $\mathcal{D}$ is a tuple (F, H, I, S, T) where F is a failure pattern, $H \in \mathcal{D}(F)$ is a failure detector history of $\mathcal{D}$, I is an initial configuration of **A**,[5] S is an *infinite* schedule of **A** that is applicable to I, and T is an *infinite* sequence of increasing time values such that, for all $i \geq 0$, if $S[i] = (p, o, v)$, then (1) p has not crashed by time $T[i]$, i.e., $p \notin F(T[i])$; (2) if $o = query$, then $v = H(p, T[i])$; and (3) every correct process in F takes an infinite number of steps in S.

The correctness of an algorithm may depend on certain aspects of the "environment" — for example, the number of processes that crash. For our purposes, an *environment* $\mathcal{E}$ is simply a set of failure patterns. For instance, $\mathcal{E}$ could be the set of all failure patterns in which no more than f processes crash, where f is some parameter.

[5] That is, a configuration in which each process is in an initial state, and the registers have the initial values specified by **A**, if any.

2.3 The Consensus Problem

In the *Consensus* problem, each process p starts with an initial value $init_p$ originally only known to itself. (Formally, this initial value is reflected in the initial state of $\mathbf{A}_p$.) Through communicating with other processes, all correct process must eventually decide (irrevocably) the same value, and that value must be one of the processes' initial values.

More precisely, an algorithm $\mathbf{C}_\mathcal{D}$ solves Consensus using a failure detector $\mathcal{D}$ in environment $\mathcal{E}$ if and only if for any failure pattern $F \in \mathcal{E}$, failure detector history $H \in \mathcal{D}(F)$, and initial configuration I of $\mathbf{C}_\mathcal{D}$, every run (F, H, I, S, T) of $\mathbf{C}_\mathcal{D}$ satisfies the following properties:

Termination: Every correct process eventually decides a value.
Validity: Every correct process decides the initial value of some process.
Agreement: No two correct processes decide different values.

3 Solving Consensus Using Failure Detectors

In this section, we present two algorithms that solve Consensus in asynchronous shared-memory systems using different types of failure detectors; the first uses a Strong failure detector, while the second uses an Eventual Strong one. In both algorithms the n processes $p_1, \ldots, p_n$ communicate through n shared 1-writer-n-reader atomic registers $r_{p_1}, \ldots, r_{p_n}$. Register r_p can be written only by p, but can be read by all processes. Both algorithms are wait-free, i.e., they can tolerate up to $n-1$ crash failures.

Besides standard programming language constructs, we use the following notation in the description of the algorithms: $\mathbf{read}(r, v)$ denotes a read operation on register r, returning value v. Similarly, $\mathbf{write}(r, v)$ denotes the operation of writing v into r. $\mathcal{S}_p$ denotes the set of suspected processes returned by p's module of a Strong failure detector. $\diamond\mathcal{S}_p$ has a similar meaning, except for an Eventual Strong failure detector.

In both algorithms, the computation of each process consists of a sequence of asynchronous rounds. (We shall be more specific about what constitutes a round as we discuss each algorithm.) We denote the value of a local variable var_p of process p *at the end* of round l as var_p^l; var_p^0 is the value of the variable just before p's round 0.

3.1 An Algorithm Using a Strong Failure Detector

In this extended abstract we present a version of the algorithm that solves *binary* Consensus — i.e., the special case where the initial values of processes are 0 or 1. A minor variation of the algorithm (with a somewhat more complex proof, however) solves the general problem. This will be given in the full version of the paper.

The code for process p

shared: r_{p_i} (initially $(0, \bot)$) for $i = 1, \ldots, n$

1 $C_p \leftarrow \{p_1, \ldots, p_n\}$
2 $v_p \leftarrow init_p$
3 **for** $l \leftarrow 1$ **to** n **do**
4   **write**$(r_p,\ (l, v_p))$
5   **repeat**
6       $M_p \leftarrow \{(q, l_q, v_q) \mid \textbf{read}(r_q, (l_q, v_q))$ where $l_q \geq l$ **and** $q \in C_p\}$
7   **until** $[\forall q \in C_p :\ (q, l_q, v_q) \in M_p$ **or** $q \in \mathcal{S}_p]$
8    $C_p \leftarrow \{q \mid (q, l_q, v_q) \in M_p\}$
9   **if** $v_p = 1$ **and** $\exists (q, l_q, 0) \in M_p$ **then**
10       $v_p \leftarrow 0$
11 **write**$(r_p,\ (n+1, v_p))$
12 **repeat**
13    $M_p \leftarrow \{(q, l_q, v_q) \mid \textbf{read}(r_q, (l_q, v_q))$ where $l_q = n + 1$ **and** $q \in C_p\}$
14 **until** $[\forall q \in C_p :\ (q, n+1, v_q) \in M_p$ **or** $q \in \mathcal{S}_p]$
15 **if** $\exists (q, n+1, 1) \in M_p$ **then**
16   decide 1
17 **else**
18   decide 0

Alg. 1. Solving Consensus using a Strong failure detector $\mathcal{S}$

An execution of Algorithm 1 proceeds in $n+1$ asynchronous rounds. Each process maintains a current "estimate" for the decision, and proposes that estimate in each round. Process p's initial proposal is its initial value $init_p \in \{0, 1\}$. A process with proposal 1 changes its proposal to 0 in any one of the first n rounds if some process that it still "trusts" (i.e., that it has not yet found being suspected by its failure detector module) has proposal 0 (lines 9–10). In the last round, $n+1$, if some correct process that is never suspected by other processes (such a process must exist by the Weak Accuracy property) has proposal 1, then all correct processes will decide 1; otherwise all correct processes will decide 0 (lines 15–18).

In Algorithm 1, each process p has three local variables v_p, C_p, and M_p that store the estimate of the decision, the set of processes that p trusts, and the current set of proposals by processes in C_p, respectively. We say that p is in round $1 \leq l \leq n$ if it is in the lth iteration of the **for** loop; it is in round $n+1$ if it has completed the **for** loop. We say that p *proposes* v in round l if $v_p^l = v$. Finally for $1 \leq l \leq n$ (i.e., for the first n rounds only), we say that p *changes its proposal* in round l if $v_p^{l-1} \neq v_p^l$. Note that by this definition a process can only change its proposal from 1 to 0 (see lines 9–10).

Lemma 1. *For any round l, $1 \leq l \leq n$, if no process changes its proposal in round l, then no process will change its proposal after round l.*

Proof: Omitted from the extended abstract. ■

Lemma 2. *For any process p and round $l \geq 1$, if p changes its proposal in round l, then there exist $l+1$ distinct processes $q_0, q_1, \ldots, q_l(=p)$ such that process q_0 has initial value 0 and q_j changes its proposal in round j, for all $j, 1 \leq j \leq l$.*

Proof: By induction on l. Consider the base case, $l = 1$. Since p changes its proposal in round 1, there must exist another process q_0 that has initial value 0. (Otherwise, all processes have initial values 1, and no process would change its proposal.)

For the induction step, assume that Lemma 2 is true for all $l \leq m$, for some $m \geq 1$. Suppose that p changes its proposal in round $m+1$. By Lemma 1, there must exist a process q that changes its proposal in round m. By induction hypothesis, we have $m+1$ distinct processes $q_0, \ldots, q_m(= q)$ with the required property. In addition, $p \notin \{q_0, \ldots, q_m\}$ as process p proposes 1 in every round before $m+1$. So, we get $m+2$ distinct processes $q_0, q_1, \ldots, q_m, p$ that satisfy the lemma, as wanted. ■

Theorem 3. *Algorithm 1 is a wait-free Consensus algorithm for asynchronous shared-memory systems with a Strong failure detector.*

Proof: By Strong Completeness, every correct process eventually suspects all processes that have crashed. So, no correct process will wait forever in the two **repeat until** statements (lines 5–7, 12–14) of Algorithm 1. Hence, every correct process will decide a value, and Termination is satisfied. Since the initial proposal of every process is its initial value, Validity is satisfied. It remains to show that Algorithm 1 also satisfies Agreement.

Let c be a correct process that is never suspected by other processes. (Process c is guaranteed to exist by the Weak Accuracy property of Strong failure detectors.) From lines 9–10 of Algorithm 1, if $init_c = 0$ then all processes that complete round 1 will have proposals 0 at the end of round 1. Note that once a process proposes 0 in a round, it will propose 0 in all subsequent rounds. Therefore, for every correct process p, if $(q, n+1, v_q)$ is in M_p^{n+1} then $v_q = 0$. Thus, all correct processes will decide 0 (lines 15–18).

Assume $init_c = 1$. If process c does not change its proposal during the execution of the **for** loop, i.e., c never executes line 10 of Algorithm 1, then all correct processes will decide 1 (lines 15–18), as $(c, n+1, 1) \in M_p^{n+1}$ for every correct process p. Suppose that process c changes its proposal in round m. There are two cases to consider:

1. $(m < n)$: All processes that complete round $m+1$ must have proposals 0 at the end of round $m+1$ (lines 9–10). Hence, for every correct process p, if $(q, n+1, v_q)$ is in M_p^{n+1}, then $v_q = 0$. Thus, in this case, all correct processes decide 0.
2. $(m = n)$: By Lemma 2, we would have $n+1$ distinct processes. This leads to a contradiction. ■

Algorithm 1 solves only binary Consensus and uses registers of size $O(\log n)$. By a slight modification, Algorithm 1 can solve multi-valued Consensus using registers of size $O(\log |V|)$, where V is the domain of the initial values. The main change is that in the first n rounds each process changes its proposal to the *minimum* of the values in M_p, while in the last round it chooses the *maximum* of the values in M_p as its final proposal (and decision). We omit this version of the algorithm from the extended abstract.

3.2 An Algorithm Using an Eventual Strong Failure Detector

We now give an algorithm that solves Consensus using an Eventual Strong failure detector. Each process p has two local variables l_p and v_p, which serve as a round counter and an estimate of the decision, respectively. We say that process p is in round l if $l_p = l$. The values of the shared registers in this algorithm are of the form (l, v, g) where l is the round number in which the value was written, v is the current proposal of the register's owner and g is a tag that could be *announce*, *propose*, or *decide*.

Algorithm 2 follows the "rotating coordinator" paradigm [6]. Process p_i is the coordinator of asynchronous round l, if and only if $i = (l \textbf{ mod } n) + 1$. In every round l, each process p first "announces" its estimate by writing $(l, v_p, \mathit{announce})$ into its own register r_p. Depending on whether it is the coordinator of round l, p does the following:

1. If p is the coordinator of round l and if no process is ahead of it (i.e., has reached a later round), then p changes its estimate to that most recently proposed by some process, if any, and "proposes" its estimate by writing $(l, v_p, \mathit{propose})$ into r_p. After p proposes its estimate, if all other processes are still in round l or below, then p decides its own estimate. Otherwise, p advances to round $l + 1$.
2. If process p is *not* the coordinator of round l, then it waits until (1) the coordinator c of round l has decided, (2) c has advanced beyond round l, or (3) its local failure detector module suspects c. In the first case, p decides c's estimate; otherwise, it advances to round $l + 1$.

In both cases, if p decides value v in round l, it writes (l, v, decide) into r_p. Note that, in Algorithm 2, procedure *scan-registers*() reads the n shared registers (in arbitrary order) and returns the set of values read, each prefixed with the name of the process that owns the register.[6] Also, Algorithm 2 uses unbounded memory as variable l_p's may grow indefinitely.

Lemma 4. *No correct process can be stuck in the* **repeat until** *statement (lines 22–24).*

Proof: Omitted from the extended abstract. ■

[6] This procedure is *not* to be confused with the *scan* of an atomic snapshot object [1]: We do not require that *scan-registers* be linearised with respect to the other write and *scan-registers* operations. Only the individual reads that are within *scan-registers* are linearised with respect to the other read and write operations.

```
The code for process p
    shared: r_{p_i} (initially ⊥) for i = 1, ..., n
1   l_p ← 0
2   v_p ← init_p
3   repeat forever
4     l_p ← l_p + 1
5     c ← p_{(l_p mod n)+1}
6     write(r_p, (l_p, v_p, announce))
7     if p = c then     (* p is the coordinator *)
8       M_p ← scan-registers()
9       if ∃(q, l_q, v_q, decide) ∈ M_p then
10        write(r_p, (l_p, v_q, decide))
11        decide v_q and halt
12      if ∀(l_q, v_q, g_q) ∈ M_p :  l_q ≤ l_p then
13        if ∃(q, l_q, v_q, propose) ∈ M_p then
14          l_max ← max{l_q | (q, l_q, v_q, propose) ∈ M_p}
15          v_p ← v_q, where (q, l_max, v_q, propose) ∈ M_p
16        write(r_p, (l_p, v_p, propose))
17        N_p ← scan-registers()
18        if ∀(q, l_q, v_q, g_q) ∈ N_p :  l_q ≤ l_p then
19          write(r_p, (l_p, v_p, decide))
20          decide v_p and halt
21    else     (* p is not the coordinator *)
22      repeat
23        read(r_c, (l_c, v_c, g_c))
24      until [ l_c > l_p or  g_c = decide or  c ∈ ◇S_p ]
25      if g_c = decide then
26        write(r_p, (l_p, v_c, decide))
27        decide v_c and halt
```

Alg. 2. Solving Consensus using an Eventual Strong failure detector $\diamond\mathcal{S}$

Lemma 5. *If a process decides, then all correct processes will decide.*

Proof: Suppose, to the contrary, that a correct process p never decides, even though some process has decided. Let l be the earliest round in which a process decides, and let q be that process and v be the value it decides. From Algorithm 2, q wrote (l, v, decide) in r_q in round l. By Lemma 4, p cannot be stuck in the **repeat until** statement. Therefore, p eventually advances to a round l' in which q is the coordinator. From lines 22–27 of Algorithm 2, process p will decide (in round l'). ■

Lemma 6. *At least one process will decide.*

Proof: Suppose, to the contrary, that no process ever decides. By Strong Completeness, there is a time t_1 after which every process that crashes is suspected by every correct process. By Eventual Weak Accuracy, there is a time t_2 after which some correct process c is never suspected by any other processes. Let t_3 be a time by which all faulty processes have already crashed. Let $t = \max\{t_1, t_2, t_3\}$ and l be the highest round to which any process has advanced at time t. Clearly, only correct processes take steps after t.

Let l_c be the earliest round that is greater than l in which c is the coordinator. By our assumption, c cannot decide in round l_c. This implies that c must have found some other correct process p that has already advanced beyond round l_c when it executes either line 12 or line 18 in round l_c. From line 24 of Algorithm 2, however, p can advance from round l_c to round $l_c + 1$ only if c has finished executing round l_c, since $g_c = \mathit{decide}$ and $c \in \diamond\mathcal{S}_p$ are always false to p in round l_c. This leads to a contradiction. ∎

Lemma 7. *No two processes decide different values.*

Proof: Let $\hat{l}$ be the earliest round in which a process decides by executing line 20 of Algorithm 2 (such a round must exist by Lemma 6 and the fact that it is not possible for all processes that decide to do so in line 11). Let p be that process, and $\hat{v}$ be its decision value. Clearly, p is the coordinator of round $\hat{l}$. We say that a process *proposes* its estimate in round l if it executes line 16 in round l. By this definition, only the coordinator of a round can propose a value in that round. First, we prove the following claim.

Claim 1 *For any round $l \geq \hat{l}$, if the coordinator of round l proposes a value v, then $v = \hat{v}$.*

Proof of Claim 1: We prove the claim by induction on l. The base case, $l = \hat{l}$, is obvious since only process p can propose a value in round l.

For the induction step, assume that the lemma is true for all l, $\hat{l} \leq l \leq m$, for some $m \geq \hat{l}$.

Suppose the coordinator c of round $m+1$ proposes v_c (in round m+1). Since p decides $\hat{v}$ in round $\hat{l}$, when p executes line 18 in round $\hat{l}$, it must find $(c, l_c, *, *)$ in $N_p^{\hat{l}}$ such that $l_c \leq \hat{l}$. In other words, p must have proposed $\hat{v}$ before c reaches round $\hat{l}+1$. Therefore, since $\hat{l}+1 \leq m+1$, c must find either $(p, \hat{l}, \hat{v}, \mathit{propose})$ or $(p, \hat{l}, \hat{v}, \mathit{decide})$ in M_c^{m+1}. If $(p, \hat{l}, \hat{v}, \mathit{decide}) \in M_c^{m+1}$, then c would have decided in round $m+1$ by executing line 11 and would not have proposed a value in round $m+1$. Therefore, we have $(p, \hat{l}, \hat{v}, \mathit{propose})$ in M_c^{m+1}. So, c must have executed lines 14–15 in round $m+1$. Let q be the process that c selects in line 14 in round $m+1$, i.e., c finds $(q, l_{max}, v_q, \mathit{propose}) \in M_c^{m+1}$. Hence, we have $v_c = v_q$.

By definition, q proposed v_q in round l_{max}. It is clear that $\hat{l} \leq l_{max}$ as $(p, \hat{l}, \hat{v}, \mathit{propose})$ is also in M_c^{m+1}. Also, since c proposes v_c $(= v_q)$ in round $m+1$, we must have $l_{max} \leq m+1$ (line 12). Since q is not the coordinator of round $m+1$, we have $l_{max} < m+1$. By induction hypothesis, as $\hat{l} \leq l_{max} < m+1$, we have $v_q = \hat{v}$. Therefore, $v_c = \hat{v}$. ∎

Suppose that some process q decides v in round l_q. There must exist a process q' (maybe q itself) that decides v by executing line 20 in some round l. By our assumption that $\hat{l}$ is the earliest round in which a process decides by executing line 20, we have $l \geq \hat{l}$. From Algorithm 2, process q' must have proposed v in round l. Therefore, by the claim above, we have $v = \hat{v}$. This shows that all processes that decide, decide the same value $\hat{v}$, as wanted. ■

Theorem 8. *Algorithm 2 is a wait-free Consensus algorithm for asynchronous shared-memory systems with an Eventual Strong failure detector.*

Proof: Lemmata 5 and 6 together give us Termination. Agreement holds by Lemma 7. From the algorithm, it is clear that a process can only propose a value that is some process's initial value. Thus Validity is also satisfied. ■

Chandra and Toueg proved that *in a message-passing* asynchronous system, there is no Consensus algorithm that uses an Eventual Strong failure detector unless a majority of processes are correct [6]. The proof is based on a partition argument: Suppose that $n/2$ of the processes have initial value 0 and $n/2$ of them have initial value 1. Further suppose that in reality all processes are correct. Initially, however, the failure detector is misbehaving (as is possible in the case of the Eventual Strong failure detector), and each process in the first group suspects each process in the second and vice-versa. Further, all messages sent between the two groups are delayed. To each group, it appears as though the processes in that group are correct and the processes in the other are all faulty. Thus, Validity and Termination require that the processes of the first group eventually decide 0, and those of the second group eventually decide 1. At this point, the failure detector modules stop misbehaving and conform to the requirement of the Eventual Strong failure detector (for example, by having each process suspect no one), and all delayed messages reach their destinations. In this way we have exhibited an execution that does not violate the assumptions of the model (in particular, the assumptions regarding the failure detector and the fact that messages are not lost), but which violates Agreement.

This sort of argument cannot be applied in the shared-memory model. The critical point is that, whereas messages may be delayed and therefore hidden for arbitrarily long, a write step cannot be hidden: Once taken, any process that subsequently reads the register "knows" that the write has occurred. Algorithm 2 takes advantage of this property of shared memory to achieve *wait-free* Consensus with an Eventual Strong failure detector. Thus, unlike the message-passing Consensus algorithm for Eventual Strong failure detectors of [6], our shared-memory algorithm does not require a majority of correct processes.

3.3 Weaker Failure Detectors

Chandra and Toueg give an algorithm which emulates a Strong failure detector from a Weak one, and an Eventual Strong failure detector from and Eventual Weak one. (The same algorithm accomplishes both emulations.) This algorithm was presented in [6] in the context of message-passing asynchronous systems, but can be easily adapted to our shared-memory setting, as follows:

Each process p maintains a shared register $suspects_p$, which contains the value returned by p's failure detector module when it was last queried, and a sequence number indicating how many times the module was queried. In addition, p maintains a local variable $output_p$, which is the present value of the local module of the failure detector that p is emulating. Process p periodically queries its local failure detector module (of a Weak or Eventual Weak failure detector), and writes the set of processes it obtained into $suspects_p$. In addition, p periodically reads the $suspects_q$ register of every process q (making sure that each process' register is read infinitely often). If the value of the sequence number of this register has changed since the last time p read it, p sets $output_p := (output_p \cup suspects_q) \setminus \{p, q\}$.

It can be shown that the set of processes that are kept in the local variables $output_p$ satisfy the properties of a Strong (Eventual Strong) failure detector, if the given failure detector is Weak (Eventual Weak). In other words, this algorithm emulates a Strong (or Eventual Strong) failure detector, given a Weak (or Eventual Weak) one. By combining this emulation with Algorithm 1 or Algorithm 2, we can obtain wait-free algorithms for Consensus in asynchronous shared-memory systems using a Weak or Eventual Weak failure detector.

4 Negative Results

In this section, we present some lower bounds and impossibility results about solving Consensus using failure detectors in asynchronous shared-memory systems.

4.1 Strong Failure Detectors

We give some negative results regarding Consensus algorithms that use a Strong failure detector. These apply, *a fortiori*, to algorithms that use an Eventual Strong failure detector.

Theorem 9. *If only 1-writer-n-reader atomic registers are allowed, then there is no wait-free Consensus algorithm for n processes using a Strong failure detector with less than n such registers.*

Proof: If there are less than n 1-writer-n-reader registers, then some process p never writes into any register. Consider a run where process p is the only process that is never suspected by other processes. Suppose that p has initial value 0 and all other processes have initial values 1. First, process p runs alone until it decides (this is possible as the algorithm is wait-free). By Validity, p decides 0. Then, let the remaining processes execute; by Termination, they all eventually decide. Since p never writes into any register, by Validity, all the remaining processes must decide 1. This, however, violates Agreement. ∎

The Termination condition of Consensus only requires every correct process to eventually decide. Suppose we further require that every correct process must eventually halt after it decides. Then, we have the following two impossibility results.

Theorem 10. *If the registers in the system can assume arbitrary (but valid) initial values, then there is no halting Consensus algorithm using a Strong failure detector, even when at most one process may crash.*

Proof: Suppose, to the contrary, that such a protocol exists. Consider a run in which a process p has initial value 1 and all other processes have initial values 0, and the registers in the system have arbitrary initial values. Also, process p never suspects any other processes, but it is always suspected by other processes. First, all processes but p run together until they decide and halt before p takes a step (such an execution is possible as we assume the protocol can tolerate one crash failure). By Termination and Validity, all processes but p must decide 0. Then, process p starts its execution and, by Termination and Agreement, it must eventually decide 0. Let v_r be the value of register r just before p starts its execution, for each register r in the system.

Consider another run in which all processes have initial values 1 and each register r has initial value v_r. As above, process p never suspects any other processes. In this run, however, process p first executes the algorithm alone. Since the algorithm is deterministic and since the registers have values as if all other processes had decided 0 and halted, process p must eventually decide 0 before other processes start their execution. This, however, violates Validity. ■

The requirement that the Consensus algorithm be halting is important in Theorem 10; without it the theorem does not hold. In the full paper we give a non-halting Consensus algorithm that uses a Strong failure detector but does not require shared memory to be initialised.

Theorem 9 gives a (tight) lower bound on the number of single-writer registers necessary for Consensus using a Strong failure detector. It seems possible that the number of registers required could be reduced if we can use multi-writer registers. Indeed, in the full paper, we show that we can solve n-process (halting) Consensus using a Strong (and even Weak) failure detector with just two n-writer-n-reader registers. On the other hand, two multi-writer registers are necessary, as the following theorem shows.

Theorem 11. *There is no wait-free halting Consensus algorithm for two processes using the Strong failure detector $\mathcal{S}$ that uses a single* 2*-writer-*2*-reader atomic register.*

The requirement that the algorithm is halting is necessary in Theorem 11: There is a non-halting Consensus algorithm for two processes that uses a single 2-writer-2-reader atomic register.

4.2 The Weakest Failure Detector

Informally, a failure detector $\mathcal{D}^*$ is the weakest for solving Consensus if, for any failure detector $\mathcal{D}$ that can be used to solve Consensus, it is possible to emulate $\mathcal{D}^*$ using $\mathcal{D}$. More precisely, for any failure detector $\mathcal{D}$ that can be used to solve Consensus in an environment $\mathcal{E}$, there exists an algorithm $\mathbf{T}_{\mathcal{D}\to\mathcal{D}^*}$

that *transforms* $\mathcal{D}$ to $\mathcal{D}^*$ in $\mathcal{E}$ in the following sense. Consider any execution of the system with a failure pattern $F \in \mathcal{E}$. Each process p uses the information provided by the failure detector $\mathcal{D}$, together with an algorithm $\mathbf{C}_{\mathcal{D}}$ that solves Consensus using $\mathcal{D}$, to maintain a local variable V_p so that the following holds: if H is the failure detector history defined by the local variables V_p, (i.e., the history so that $H(p,t)$ is the value of V_p at time t), then H is in $\mathcal{D}^*(F)$. Since we can emulate $\mathcal{D}^*$ using $\mathcal{D}$, any problem that can be solved using $\mathcal{D}^*$ can also be solved using $\mathcal{D}$, and so $\mathcal{D}^*$ is weaker than $\mathcal{D}$. We write $\mathcal{D} \succeq_{\mathcal{E}} \mathcal{D}^*$ to denote the fact that $\mathcal{D}$ can be used to emulate $\mathcal{D}^*$ in $\mathcal{E}$.

In [5], Chandra, Hadzilacos and Toueg show that the Eventual Weak failure detector $\diamond\mathcal{W}$ is the weakest that can be used to solve Consensus in asynchronous message-passing systems. we have shown that this result also holds for asynchronous *shared-memory* systems:

Theorem 12. *The Eventual Weak failure detector $\diamond\mathcal{W}$ is the weakest failure detector that can used to solve Consensus in asynchronous shared-memory systems.*

The proof of this theorem follows closely that in [5]. Some technical difficulties arise due to the fact that in shared-memory systems a write step of a process cannot be "hidden" from other processes (in contrast, as we have remarked, in message-passing systems, the sending of a message can be hidden for arbitrarily long by delaying that message). This proof is quite long and is therefore omitted from the extended abstract.

5 Conclusion

In Theorem 9 of Section 4, we show that if only 1-writer-n-reader registers are allowed, any wait-free Consensus algorithm for n processes that uses a Strong failure detector must use at least n such registers. Since our algorithms only use this number of registers, this result is tight. It is conceivable, however, that the number of registers required can be reduced if *multi-writer* registers can be used. We have some partial results regarding this issue, although we are far from a complete answer. On the positive side, we can show that with a Strong (or Weak) failure detector *two* n-writer-n-reader registers are sufficient (and necessary) for solving n-process halting Consensus, for arbitrary n. On the negative side, we can show that with an *Eventual* Strong (or Weak) failure detector, *three* 3-writer-3-reader registers are necessary (and sufficient) for wait-free halting Consensus among three processes. We do not know how this result generalises beyond the special case of $n = 3$. Using the technique of Fich, Herlihy and Shavit [8], it can be shown that any wait-free halting Consensus algorithm using an Eventual Weak failure detector for n processes must use at least $\Omega(\sqrt{n})$ shared n-writer-n-reader registers. We do not know whether this lower bound is tight.

Another interesting question regarding the number of registers is whether there is a trade-off between this resource and the degree of fault-tolerance of the Consensus algorithm (measured as the number of crashes that it tolerates). Our results on the number of registers required are for wait-free algorithms. It is conceivable that the number of registers needed is a function of the number of processes that can crash.

In addition to the number of shared registers, we may also be interested in the *size* of those registers. Our Consensus algorithm that uses an Eventual Strong failure detector (Algorithm 2) requires registers of unbounded size. We would like to determine whether this is necessary, or whether an algorithm can be devised that uses registers of bounded size.

References

1. Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt and Nir Shavit. Atomic snapshots of shared memory. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pages 1–13, August 1990.
2. Yehuda Afek, David S. Greenberg, Michael Merritt, and Gadi Taubenfeld. Computing with faulty shared memory. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pages 47–58, August 1992.
3. James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11:441–461, 1990.
4. Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocol. In *Proceedings of the 2th ACM Symposium on Principles of Distributed Computing*, pages 27–30, August 1983.
5. Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, August 1992. Also technical report, Department of Computer Science, Cornell University, 1993, Ithaca, NY 14853-7501.
6. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for asynchronous systems. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, August 1991.
7. Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.
8. Faith Fich, Maurice Herlihy, and Nir Shavit. On the space complexity of randomized synchronization. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, pages 241–249, August 1993.
9. Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the Association for Computing Machinery*, 32(2):374–382, April 1985.
10. Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, 1992.
11. Michael C. Loui and Hosame H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. In *Advances in Computer Research*, volume 4, pages 163–183. JAI Press Inc., 1987.

This article was processed using the LaTeX macro package with LLNCS style