Brief Announcement: The Recoverable Consensus Hierarchy

Wojciech Golab*
University of Waterloo, Department of Electrical and Computer Engineering
Waterloo, Ontario, Canada
wgolab@uwaterloo.ca

ABSTRACT

Herlihy's consensus hierarchy ranks the power of various synchronization primitives for solving consensus in a model where asynchronous processes communicate through shared memory, and may fail by halting. This paper revisits the consensus hierarchy in a model with crash-recovery failures, where the specification of consensus, called *recoverable consensus* in this paper, is weakened by allowing non-terminating executions when a process fails infinitely often. Two variations of this model are considered: with independent process failures, and with simultaneous (i.e., system-wide) process failures. We prove two fundamental results: (i) Test-And-Set is at level 2 of the recoverable consensus hierarchy if failures are simultaneous, and similarly for any primitive at level 2 of the traditional consensus hierarchy; and (ii) Test-And-Set drops to level 1 of the hierarchy if failures are independent, unless the number of such failures is bounded. To our knowledge, this is the first separation between the simultaneous and independent crash-recovery failure models with respect to the computability of consensus.

CCS CONCEPTS

• **Theory of computation → Shared memory algorithms**.

KEYWORDS

concurrency; shared memory; consensus; fault tolerance; theory

ACM Reference Format:
Wojciech Golab. 2019. Brief Announcement: The Recoverable Consensus Hierarchy. In *2019 ACM Symposium on Principles of Distributed Computing (PODC'19), July 29–August 2, 2019, Toronto, ON, Canada.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3293611.3331574

1 INTRODUCTION

Herlihy's consensus hierarchy [4] ranks the power of synchronization primitives for solving consensus – a problem where processes agree on a decision chosen from a set of proposed values. The position of a primitive P in the hierarchy is defined by its *consensus number* – the largest n such that P used in conjunction with read/write registers solves n-process consensus in the asynchronous shared memory model with permanent crash failures. Test-And-Set (TAS) is at level two of this hierarchy as it can be used to solve 2-process (but not 3-process) consensus: a TAS object is initialized to 0, each process then announces its proposal in a shared read/write register, executes a TAS operation, returns its own proposal if it wins, and returns the proposal of the other process if it loses. Motivated by the possibility of recovering program state from non-volatile main memory (NVRAM) after a failure, we revisit the consensus hierarchy in a *crash-recovery* model where a failure merely resets the private variables of a process (including its program counter) to their initial values, and preserves the state of shared variables.

Building on the results of Berryhill, Golab, and Tripunitara [2], who proved that consensus remains sufficiently powerful to implement any shared object type for any number of processes in the crash-recovery failure model, we consider solutions of the *recoverable consensus* (RC) problem. RC is a natural adaptation of consensus to crash-recovery failures that relaxes wait-freedom by allowing non-terminating executions in cases where failures occur infinitely often. The correctness properties of RC are defined as follows:

(1) *Agreement:* distinct processes never output different decisions.
(2) *Validity:* each decision returned is the proposal value of some process.
(3) *Recoverable wait-freedom:* each time a process executes its algorithm from the beginning, it either returns a decision after a finite number of its own steps, or crashes.

The *recoverable consensus hierarchy* is the analog of the traditional consensus hierarchy for the RC problem in the crash-recovery model, and defines (recoverable) consensus numbers for a variety of shared object types and synchronization primitives. We establish two fundamental results with respect to the position of TAS in this new hierarchy: (i) TAS is at level 2 if failures are simultaneous, and similarly for any primitive at level 2 of the traditional consensus hierarchy; and (ii) TAS drops to level 1 if failures are independent, unless the number of such failures is bounded. Intuitively, our result captures the observation that when failures are simultaneous, a process recovers with more information regarding the states of other processes than when failures are independent. To our knowledge, this is the first separation between the simultaneous and independent crash-recovery failure models with respect to the computability of consensus.

2 RESULTS FOR SIMULTANEOUS FAILURES

This section presents a technique for transforming any 2-process conventional consensus algorithm into a 2-process RC algorithm that tolerates arbitrarily many simultaneous crash-recovery failures. The transformation is presented in detail in Figure 1. It uses

*Author supported in part by an Ontario Early Researcher Award, and by a Google Faculty Research Award.

PODC '19, July 29–August 2, 2019, Toronto, ON, Canada

ACM ISBN 978-1-4503-6217-7/19/07…$15.00
https://doi.org/10.1145/3293611.3331574

Shared variables:

- $P[1..2]$: **array** of proposal values, **init** $\perp$
- C: conventional wait-free 2-process consensus object
- D: decision, **init** $\perp$

Private variables:

- other: process ID
- d: decided value

Procedure Decide(v: proposal value) for proc. p_i, $i \in 1..2$

```
1  if i = 1 then other := 2 else other := 1
2  if P[i] = ⊥ ∧ P[other] = ⊥ then
3  |  P[i] := v
4  |  d := C.Decide(v)
5  |  D := d
6  |  return d
7  else if D ≠ ⊥ then
8  |  return D
9  else if P[i] ≠ ⊥ ∧ P[other] = ⊥ then
10 |  return P[i]
11 else if P[i] = ⊥ ∧ P[other] ≠ ⊥ then
12 |  return P[other]
13 else // P[i] ≠ ⊥ ∧ P[other] ≠ ⊥
14 |  return P[1]
```

Figure 1: Transformation from 2-process conventional consensus to 2-process recoverable consensus.

a shared array $P[1..2]$ to announce proposals, a conventional 2-process consensus algorithm C to reach agreement in some scenarios (e.g., failure-free executions), and a shared variable D to record the decision value computed using C.

Starting from the initial state where both elements of $P[1..2]$ are initialized to $\perp$, process p_i records its proposal, executes C, and records the outcome in D (lines 2–6). Assuming that some process completes line 5, recovery from a failure is achieved easily by returning the value saved in D (lines 7–8). However, recovery from a failure prior to line 5 is more difficult because if the failure occurred while some process p_i was at line 4 then the algorithm must guarantee that p_i does not access C incorrectly (i.e., by resuming the interrupted execution of C.**Decide** from the beginning). Recovery in this scenario is accomplished by case analysis. If a failure occurs before any process has completed line 3, then the execution path on recovery is identical to the failure-free path since $P[1..2]$ and C remain in their initial states. On the other hand, if some process did complete line 3, then the algorithm terminates after a bounded number of read operations, without updating shared memory. If exactly one element of $P[1..2]$ is $\perp$ (lines 9–12) then the algorithm returns the unique recorded proposal. Finally, if both elements of $P[1..2]$ are non-$\perp$, then the algorithm makes an arbitrary but deterministic choice among these two proposal values and returns the chosen one (lines 13–14).

The correctness properties of the algorithm are captured in Theorem 2.1, whose detailed proof appears in [3].

Theorem 2.1. *The algorithm shown in Figure 1 satisfies agreement, validity, and recoverable wait-freedom.*

Because the non-recoverable consensus algorithm C can be implemented in a wait-free manner using TAS, as explained in Section 1, Theorem 2.1 implies that TAS is at level 2 (or higher) of the RC hierarchy if failures are simultaneous.

3 RESULTS FOR INDEPENDENT FAILURES

3.1 Solution for Finitely Many Failures

This section presents a technique for transforming any n-process conventional consensus algorithm into an n-process recoverable consensus algorithm that tolerates up to a (predefined) number f of independent crash-recovery failures. The transformation is presented in detail in Figure 2, and uses $f + 1$ instances of the conventional consensus algorithm denoted by the array $C[0..f]$. To a first approximation, the transformation works by having each process p_i access the $f + 1$ consensus algorithms in a for loop at line 15 until the **Decide** procedure is executed to completion without failing. The array $R[1..n]$ is used at line 16 and line 17 to determine which consensus algorithm in the array $C[0..f]$ will be accessed in the next iteration by each process, and hence to avoid unsafe access to these base objects. Assuming that there are at most f failures, this strategy ensures that p_i eventually computes a decision because the total number of iterations required is at most $f + 1$. The main technical challenge lies in ensuring agreement in cases when processes compute decisions in different iterations, using distinct instances of the conventional consensus algorithm. This is accomplished by a pair mechanisms working in synergy.

In the first mechanism, a process p_i that is executing iteration k of the outer for loop checks at lines 18–20 whether a decision was reached in a lower-numbered iteration using $C[k']$ for some $k' < k$, and recorded in $D[k']$ at line 22, before p_i proceeds to execute the consensus algorithm $C[k]$ at line 21. If $D[k']$ holds such a decision value then p_i adopts this value in lieu of its own proposal at line 20. This statement is inside the inner for loop and may be executed multiple times in one iteration of the outer for loop, in which case p_i adopts the decision value corresponding to the largest possible k'. This mechanism alone is not sufficient, however, since a race can occur between a process that is about to write $D[k-1]$ and a process that is about to access $C[k]$.

The second mechanism deals with the above race condition at lines 23–26 by inspecting the elements of array $R[1..n]$ belonging to other processes. If p_i finds some element $R[z]$, $z \neq i$, holding an integer larger than $R[i]$, then p_z is at least one iteration ahead of p_i. In this case p_i "forgets" the decision it computed earlier at line 21 by resetting the private variable d at line 26, and continues to the next iteration of the for loop; the conditional statement at line 27 bypasses the return statement at line 28. If p_i does not find such a process p_z, then p_i reaches line 28, where it returns the decision it computed in the current iteration of the outer for loop. The two mechanisms combined ensure agreement despite the possibility that the consensus algorithms $C[0..f]$ may not all reach the same decision.

The correctness properties of the algorithm are captured in Theorem 3.1, whose detailed proof appears in [3].

Shared variables:

- $R[1..n]$: **array** of read/write register, **init** 0
- $C[0..f]$: **array** of conventional wait-free n-process consensus objects
- $D[0..f]$: **array** of read/write register, **init** $\perp$

Private variables:

- k, k': integers, uninitialized
- d: decision value, uninitialized

Procedure Decide(v: proposal value) for proc. p_i, $i \in 1..n$

```
15 for k in 0..f do
16     if R[i] = k then
17         R[i] := k + 1
           // check for a decision in a
              lower-numbered iteration
18         for k' ∈ 0..(k − 1) do
19             if D[k'] ≠ ⊥ then
20                 v := D[k']
21         d := C[k].Decide(v)
22         D[k] := d
           // check for a collision with a
              higher-numbered iteration
23         if k < f then
24             for z ∈ 1..n, z ≠ i do
25                 if R[z] > R[i] then
26                     d := ⊥
           // return decision if known
27         if d ≠ ⊥ then
28             return d
```

Figure 2: Transformation from n-process conventional consensus to n-process recoverable consensus ($\leq f$ failures).

Theorem 3.1. *The algorithm shown in Figure 2 satisfies agreement, validity, and recoverable wait-freedom in every execution with at most f failures.*

3.2 Impossibility Result for Test-And-Set

Any impossibility result for solving consensus in the conventional asynchronous model with halting failures (or without failures) applies also to solving RC in the asynchronous model with crash-recovery failures. This is because any execution that is possible in the conventional model is also admissible in the crash-recovery model, and because a violation of wait-freedom in such an execution implies a violation of recoverable wait-freedom as well. Thus, the position of a primitive in the RC hierarchy cannot be higher than its position in the traditional consensus hierarchy [4]. In particular, TAS can be no higher than at level 2 in the RC hierarchy, and so the analysis from Section 2 implies that TAS is precisely at level 2 for simultaneous failures. In this section, we extend and complete the latter result by settling the position of TAS in the RC hierarchy for independent crash-recovery failures.

Theorem 3.2. *The 2-process RC problem cannot be solved using readable Test-And-Set objects and read/write registers if failures are independent.*

Proof sketch. We suppose for contradiction that such a solution does exist. The proof uses a valency argument inspired by Herlihy's [4] and adapted to work in the crash-recovery model.[1] A restricted subset of executions is considered where only one designated process p_1 may crash, and only in states satisfying two criteria: (i) p_1's previous step was the first TAS operation by p_1 on some TAS object; and (ii) the same TAS object has been accessed by p_2 as well. Following the standard argument, processes p_1 and p_2 take steps until a bivalent state s is reached where every enabled step is a decision step (i.e., leads to a univalent state). Recoverable wait-freedom ensures that this construction terminates eventually, as otherwise p_1 crashes infinitely often, in which case p_2 is stuck forever without crashing because each crash by p_1 is associated with a distinct step by p_2. Moreover, p_1 and p_2 are enabled to take decision steps leading to a v_1-valent and v_2-valent state, respectively, for some $v_1 \neq v_2$. These decision steps must be TAS operations on the same shared object that is in its initial state of 0, which implies that p_1 has not yet applied a TAS to this object. If p_1 takes its decision step followed by p_2, and then p_1 crashes, we arrive at a v_1-valent state s_1. Similarly, if p_2 takes its decision step followed by p_1, and then p_1 crashes, we arrive at a v_2-valent state s_2. The states s_1 and s_2 are indistinguishable to p_1, and a contradiction is then reached following the standard argument since $v_1 \neq v_2$. □

The extended version of this paper [3] uses an alternative form of the above proof to relate the number of TAS objects used by a recoverable consensus algorithm to the maximum number of failures it can tolerate:

Theorem 3.3. *For any integer $f > 0$, there is no algorithm that uses at most f readable Test-And-Set objects and any number of read/write registers, and solves 2-process recoverable consensus in executions with up to f independent failures.*

Theorem 3.3 implies that the transformation shown in Figure 2 is optimal as it can be instantiated to use $f+1$ TAS-based conventional 2-process consensus objects to solve 2-process recoverable consensus in executions with up to f independent failures.

REFERENCES

[1] Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Nesting-safe recoverable linearizability: Modular constructions for non-volatile memory. In *Proc. of the 2018 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 7–16, 2018.

[2] Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In *Proc. of the 19th International Conference on Principles of Distributed Systems (OPODIS)*, pages 20:1–20:17, 2016.

[3] Wojciech Golab. Recoverable consensus in shared memory. *CoRR*, 2018. URL https://arxiv.org/abs/1804.10597.

[4] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[1] Attiya, Ben-Baruch, and Hendler developed a similar proof technique in parallel with our work, in the context of *Nesting-Safe Recoverable Linearizability* [1]. They use a more specialized notion of valency that captures the possible outputs of an algorithm in *failure-free* execution fragments. A state that is univalent in their analysis can be either univalent or bivalent according to our definition.