NVRAM - response recover linearzability [*]

Hagit Attiya[1], Ohad Ben-Baruch[2], and Danny Hendler[3]

[1]Department of Computer-Science, Technion, hagit@cs.technion.ac.il
[2]Department of Computer-Science, Ben-Gurion University, ohadben@post.bgu.ac.il, +972(0)524261187 [†]
[3]Department of Computer-Science, Ben-Gurion University, hendlerd@cs.bgu.ac.il, +972(0)86428038

February 14, 2018

Abstract

abstract goes here...

[*]Partially supported by the Israel Science Foundation (grants 1227/10, 1749/14) and by the Lynne and William Frankel Center for Computing Science at Ben-Gurion University.

[†]Contact author.

1 Introduction

Shared-memory multiprocessors are now prevalent anywhere from high-end server machines, through desktop and laptop computers to smartphones, accelerating the shift to concurrent, multi-threaded software. Concurrent software involves a collection of threads, each running a separate piece of code, possibly on a different core. Since threads may be delayed because of a variety of reasons (such as cache-misses, interrupts, page-faults, and scheduler preemption), shared-memory multiprocessors are *asynchronous* in nature.

Asynchrony is also related to reliability, since asynchronous algorithms that provide nonblocking progress properties (e.g., lock-freedom and wait-freedom) in an asynchronous environment with reliable processes continue to provide the same progress properties when *crash failures* are introduced. Informally speaking, this property holds because a process that crashes permanently at an arbitrary point in the execution of its algorithm is indistinguishable to the other processes from one that is merely very slow. Owing to its simplicity and intimate relationship with asynchrony, the crash-failure model is almost ubiquitous in the treatment of non-blocking shared memory algorithms.

The attention to the crash-failure model has so far mostly neglected the *crash-recovery* model, in which a failed process may be resurrected after it crashes. The reason is that the crash-recovery model is poorly matched to multi-core architectures with volatile SRAM-based caches and DRAM-based main memories: Any state stored in main memory is lost entirely in the event of a system crash or power loss, and recording recovery information in non-volatile secondary storage (e.g., on a hard disk drive or solid state drive) imposes overheads that are unacceptable for performance-critical tasks, such as synchronizing threads inside the operating system kernel.

This separation between volatile main memory and non-volatile secondary storage has led to a partitioning of the program state into operational data stored using in-memory data structures, and recovery data stored using sequential on-disk structures such as transaction logs. In the event of a system-wide failure, such as a power outage, in-memory data structures are lost and must be reconstructed entirely from the recovery data, making the software system that relies on them temporarily unavailable. As a result, the design of in-memory data structures emphasizes disposable constructs optimized for parallelism, in contrast to the structures that hold recovery data, which cannot be discarded upon a failure and which benefit less from parallelism since their performance is limited by the secondary storage.

Recent developments in non-volatile main memory (NVRAM) media foreshadow the eventual convergence of primary and secondary storage into a single layer in the memory hierarchy, combining the performance benefits of conventional main memory with the durability of secondary storage. Traditional log-based recovery techniques can be applied correctly in such systems but fail to take full advantage of the parallelism enabled by allowing processing cores to access recovery data directly using memory operations rather than slow block transfers from secondary storage.

In this paper we investigate how the performance benefits of NVRAM can be harnessed for improving the robustness of shared-memory programs by allowing concurrent algorithms to efficiently recover from crash failures. This challenge requires a careful rethinking of recovery mechanisms and involves addressing both foundational and algorithmic research questions. Our emphasis would be on direct, non-transactional, approaches of implementing algorithms for *recoverable concurrent* (also called *persistent*) objects. More specifically, we formulate an abstract model of NVRAM shared-memory multiprocessors, and discuss known correctness and progress conditions, while proposing a new property called *recoverable-response linearizability.* We then discuss the theoretic limitations of these systems, on the one hand, and construction of concurrent algorithms that support effective recovery from crash failures, on the other hand. We focus on construction of recoverable versions of primitive memory operations such as read, write, swap, compare-and-swap and fetch-and-add, deriving upper and lower bounds, and how these recoverable versions can be used to implement various applications.

1.1 Related work

When an application implements a concurrent data structure, it is necessary to specify its *semantics*, namely, the properties provided by values it returns, which determine the feasibility and complexity of implementing it. The *correctness* condition of a concurrent data structure specifies how to derive the semantics of concurrent implementations of the data structure from the corresponding *sequential specification* of the data structure. This requires to disambiguate the expected results of concurrent operations on the data structure.

Two common ways to do so are *sequential consistency* [14] and *linearizability* [10]. Both require that the values returned by the operations appear to have been returned by a sequential execution of the same operations; sequential consistency only requires this order to be consistent with the order in which each individual thread invokes the operations, while linearizability further requires this order to be consistent with the real-time order of non-overlapping operations. The standard shared-memory model assumes that shared memory is linearizable or at least sequentially consistent.

Linearizability is ill-equipped to specify the correctness criteria for implementations that support crash-recovery failures, since linearizability has no notion of an aborted or failed operation, and requires that a process finish one operation before it invokes the next. Extended versions of linearizability or, more generally, alternative definitions are required for specifying correctness for such implementations.

Aguilera and Frølund [1] proposed *strict linearizability* as a correctness condition for persistent concurrent objects, which treats the crash of a process as a response, either successful or unsuccessful, to the interrupted operation. A successful response means that the operation takes effect at some point between its invocation and the crash failure, and an unsuccessful response means that the operation does not take effect at all. Strict linearizability preserves both *locality* [9] and program order. Although this property seems like a natural extension for linearizability, Aguilera and Frølund proved that unlike linearizability it precludes the implementation of some wait-free objects for certain machine models: there is no wait-free implementations of multi-reader single-writer (MRSW) registers from single-reader single-writer (SRSW) registers under strict linearizability.

Guerraoui and Levy [8] proposed *persistent atomicity.* It is similar to strict linearizability, but allows an operation interrupted by a failure to take effect before the subsequent invocation of the same process, possibly after the failure. They also proposed *transient atomicity*, which relaxes this criterion even further and allows an interrupted operation to take effect before the subsequent write response of the same process. Both conditions ensure that the state of an object will be consistent in the wake of a crash, but they fail to provide locality: correct histories of separate objects, when merged, will not necessarily yield a correct composite history.

Guerraoui and Levy properties capture, in some sense, the flexibility of implementations which uses helping mechanism, where an interrupted operation by some failed process p can be completed by a different process after p crashes. Hence, we would like to have the option to set the linearization point after the crash of p, where strict linearizability does not allow so. Moreover, Censor-Hillel et al. [3] formalised the notion of helping, and proved that some objects, when implemented wait-free, must use an helping mechanism.

In addition, it is not clear how to generalize transient atomicity to a general object which does not support a write operation, as the definition explicitly uses write operation in order to determine the allowed interval for the linearization point. Moreover, transient atomicity allows a scenario where a process p crash during an operation on object X, then recovers and complete a read operation on X, while the linearization point of the interrupted operation is set to after the read. That is, we get a history where p executes two overlapping operations on the same object, causing a program order inversion, and violets the well-formedness property (defined in Section...), which is in the heart of the linearizability definition.

In order to overcome the above drawbacks Berryhill et al. [2] proposed an alternative condition, called *recoverable linearizability*, which achieves locality but may sacrifice program order after a crash. It is a relaxed version of persistent atomicity, which requires the operation to be linearized or aborted before any subsequent linearization by the pending thread on that same object.

Izraelevitz et al. [11] considered a real-world failure model, in which processes are assumed to fail together, as part of a full-system crash. Under this model, persistent atomicity and recoverable linearizability are indistinguishable (and thus local). The term *durable linearizability* was used to refer to this merged consistency condition under the restricted failure model.

The goal of all correctness conditions defined above for the crash-recovery model is to maintain the state of concurrent objects consistent in the face of crash failures. However, to the best of our knowledge, none of them guarantees that the recovery code is always able to infer whether the failed operation completed and, if so, to obtain its response value. Inferring this may be challenging, since function responses are returned via volatile processor registers.

As a simple example, consider a base object $\mathcal{B}$ that supports the *read* and *write* operations. Suppose that a process p crashes while performing a $\mathcal{B}.write(v)$ operation, which, upon completion, should returns *ack*. Since the write is atomic, there is no problem in ensuring the consistency of $\mathcal{B}$ in spite of p's failure: either the write took effect before the failure, or it did not. However, in this case, once the process recovers from the failed *write* operation it has no way of knowing whether the write occurred or not. This is because p cannot tell whether the failure occurred before v was written to NVRAM or whether it occurred after v was written to NVRAM (and was possibly then overwritten) but the *ack* value, written to a volatile local variable, was lost because of the failure. Without this knowledge, if the failed *write* operation on $\mathcal{B}$ was applied by an operation of another recoverable object $\mathcal{O}$, $\mathcal{O}$ may not be able to recover correctly.

To address this problem, we consider the model proposed by Golab and Ramaraju for investigating mutual exclusion [7] and later used by [6, 12], which is the following. A set of asynchronous processes communicate by accessing atomic *non-volatile shared-memory variables.* In addition, each process has *volatile private variables* stored in processor registers. *Processes are unreliable* in the following sense: at any point in time, while executing a concurrent object function, a process p may incur a crash failure, causing all its local variables to be reset to arbitrary values.

A concurrent object is called *recoverable*, if it implements a `Recover` function that is responsible for fixing its internal state following a failure and allows it to either correctly complete its failed operation and obtain its response value, or re-try it, so it may proceed to perform its next operation. If the failure occurs while p executes an operation of a recoverable object, then, upon resurrecting p, the application or operating system guarantees that p's execution proceeds by executing the object's `Recover` function. It is assumed that the `Recover` function has access to the value of p's program counter at the time of the failure. *We note that knowledge of the last PC value leaves* *uncertainty regarding whether that last instruction was performed or not and, if it was, what its return value was.*

We propose a new correctness property called *recoverable response linearizability* (RR-linearizability). Informally, RR-linearizability allows the recovery code to 'extend' the interval of the failed operation until the end of the recovery code. It requires a transactional effect guaranteeing that either the operation is linearized at some point between its invocation and the end of recovery code (which may attempt to complete it), or the operation has no effect. In addition, unlike previous definitions, RR-linearizability requires also that, following recovery, process p is able to determine whether the operation was linearized or not and, if it was, what its response value is.

2 Model and Definitions

2.1 Standard Shared-Memory Model

We use a standard model, based on Herlihy and Wing's model [10], of an asynchronous shared memory system. A set P of $n > 1$ *processes* $p_0, \ldots, p_{n-1}$ communicate by applying operations on shared *base objects* that support atomic operations, e.g., reads, writes and read-modify-write, to shared variables. No bound is assumed on the size of a shared variable (i.e., the number of distinct values it can take). Base objects are used in order to construct more complex implemented objects, such as queues and stacks, by defining access procedures that simulate each operation on the implemented object using operations on base objects.

The interaction of processes with implemented objects is modelled using steps and histories. There are four types of steps: (1) an invocation step, denoted (INV, p, X, op), represents the invocation by process p of an operation op on implemented object X; (2) a response step, denoted (RES, p, X, ret), represents the completion by process p of the last operation it invoked on object X, with response ret; (3) a crash step, denoted $(CRASH, p)$, represents the crash of a processes p;

A *history* H is a sequence of steps. Given a history H, we use $H|p$ to denote the subhistory of H containing all and only the events performed by process p. Similarly, $H|O$ denotes the subhistory of H containing all and only the events performed on object O, plus crash and recovery events. A response step is *matching* with respect to an invocation step s by a process p on object X in a history H if it is the first response step by p on X that follows s in H, and it occurs before p's next invocation (if any) in H.

Given a history H and a process p, an *operation* by p in H comprises an invocation step and its matching response, if it exists. An operation is *complete* if it has a matching response step, and *pending* otherwise. Given two operations op_1 and op_2 in a history H, we say that op_1 *happens before* op_2, denoted by $op_1 <_H op_2$, if op_1 has a matching response that precedes the invocation step of op_2 in H. If neither $op_1 <_H op_2$ nor $op_2 <_H op_1$ holds then we say that op_1 and op_2 are *concurrent* in H.

A history H is *sequential* if no two operations in it are concurrent. Two histories H and H' are *equivalent* if for every process p, $H|p = H'|p$ holds. A history H is *well-formed* if for each process p, each invocation step in $H|p$ is immediately followed by a matching response, or by a crash step, and every response step in $H|p$ is a matching response for a preceded invocation. Informally, H is well-formed if $H|p$ is a sequential history of operations, except for the ones that may not have a response step due to crash steps.

An object O is defined using a *sequential specification* which defines its allowed behaviors and is expressed as a set of possible sequential histories over O. A sequential history H is legal if for every implemented object O accessed in H, $H|O$ belongs to the sequential specification of O.

2.1.1 Correcntess Conditions

We now consider different variants of correctness conditions for NVRAM systems which takes into consideration failures. We follow Berryhill et al. [2] for formal definitions. For each variant, given a history H, we first define a way to extend H such that some pending operations are supplied with a matching response, and the rest pending operations are removed, followed by a definition containing requirements from the resulted extension.

All the following variants are in some sense a natural extension for the Herlihy and Wing's linearizability property, since it is widely used in conventional shared memory models. As such, we first give a formal definition for linearizability. However, linearizability does not support crash steps, and therefore the definition is valid for history H which is free of crash steps. Given such a history H, a *completion* of H is a history H' constructed from H by appending matching responses for a subset of pending operations, and then removing any remaining pending operations.

Definition 1 (Linearizability)**.** *A finite history H is linearizable if it has no crash events, and it has a completion H' and there exists a legal sequential history S such that:*

L1. H' is equivalent to S; and

L2. $<_H \subseteq <_S$ (i.e., if $op_1 <_H op_2$ and both ops appear in S then $op_1 <_S op_2$).

For strict linearizability, a *strict completion* of H is a history H' constructed from H by inserting matching responses for a subset of pending operations after the operations invocation and before the next crash step (if any), and finally removing any remaining pending operations and crash steps.

Definition 2 (Strict linearizability)**.** *A finite history H is strictly linearizable if it has a strict completion H' and there exists a legal sequential history S such that:*

SL1. H' is equivalent to S; and

SL2. $<_{H'} \subseteq <_S$ (i.e., if $op_1 <_{H'} op_2$ and both ops appear in S then $op_1 <_S op_2$).

For persistent linearizability, a *persistent completion* of H is a history H' constructed from H by inserting matching responses for a subset of pending operations after the operations invocation and before the next invocation step of the same process, and finally removing any remaining pending operations and crash steps.

Definition 3 (Persistent linearizability)**.** *A finite history H is persistently linearizable if it has a persistent completion H' and there exists a legal sequential history S such that both conditions SL1 and SL2 of definition 2 holds.*

For recoverable linearizability, a *recoverable completion* H' is obtained from H in exactly the same manner as a strict completion. In addition, the *invoked before* relation over a history H, denoted $\ll_H$ is an extension of the "happens before" relation, such that $op_1 \ll_H op_2$ if $op_1 <_H op_2$ or that both operations invoked by the same process p on the same object X, and the invocation step of op_1 precedes the invocation step of op_2 in H. Notice that the extension takes into account pending operations, while $<_H$ is not defined in such a case.

Definition 4 (Recoverable linearizability)**.** *A finite history H is recoverable linearizable if it has a recoverable completion H' and there exists a legal sequential history S such that:*

RL1. H' is equivalent to S; and

RL2. $\ll_H \subseteq <_S$ (i.e., if $op_1 \ll_H op_2$ and both ops appear in S then $op_1 <_S op_2$).

As shown by Berryhill et al. [2], the requirement for a strict completion H' does not prevent an operation from taking effect after a crash that interrupts it. This follows from the fact that unlike strict linearizability, we do not ask the sequential history S to respect the order $<_{H'}$, but rather the order $<_H$. For an operation *op* that was interrupt by a crash, in $<_H$ it is after any operation that was complete before the invocation of *op*, but there is no operation following *op* in $<_H$, since it has no response. Therefore, in S we allow to place *op* anywhere after its invocation without violating $<_H$. However, in order to prevent program order inversion of the same process on the same object, $\ll_H$ restrict S not place *op* in such a reverse order.

2.1.2 Recoverable Response Linearizability

As discussed in the introduction, none of the above definitions guarantee a failing process can complete its pending operation upon recovery, or at least have an access to the response value of the operation in case it is linearized. In some cases, a process might be able to know whether the operation took effect. Friedman et al. [5] used the term *detectable execution* for an implementation which satisfy this condition. Quoting from [5]: "Durable linearizability does not provide a mechanism to determine whether an operation that executed concurrently with a crash was eventually executed. Without the ability to distinguish completed operations from lost operations, it would be difficult to recover the entire program, because in practice it is often important to execute each operation exactly once."

A concurrent object is called *recoverable* if any of its operations is equipped with a `Recover` function, such that if a process crash while executing an operation on the recoverable object, upon recovering the appropriate `Recover` function is triggered (by the system), and we require the process to complete its pending operation before invoking the next one. As we explain later, the completion requirement, although seems too restrictive, does not rule out an option for the `Recover` function to abort the pending operation, as in such case the process can reissue it. Nevertheless, this restriction simplifies the definition and proofs.

Recoverable object by its own does not consider the response value of the operation. For example, a primitive CAS is a recoverable object (with an empty `Recover` function), although a process crashing after executing CAS have no access to the response value upon recovery, as it was lost. For this reason we extend the definition of recoverable object, such that in addition to a `Recover` function it also needs to satisfy the following: every operation returns (i.e., there is a response step in the history) only after the response value is persistent. In a more formal way, a process p have a designated variable Res_p in the non-volatile memory such that at the time of (RES, p, X, ret) step, the value ret is written in Res_p. Notice that the object's semantic does not change, that is, we do not require the response value to be persistent at the linearization point.

In order to formally capture this behaviour we introduce a recover step, denoted $(RECOVER, p)$, represents the recovery of a process p, and the invocation of the `Recovery` function matching the pending operation, if there is one. A history H is called *recoverable well-formed* if every crash step by a process p which is not the last step of p, is followed by a recover step of p (and no other steps of p are allowed in between), and vice verse. In addition, every response step follows either the matching invocation step or a recover step. We care only about such histories, as we assume the system always triggers the proper `Recovery` function whenever a process is waking-up after a crash.

A recoverable object is in some sense "fail-resistant". If a process crash after completing an operation, then upon recovery the process have an access to the response value residing in the non-volatile memory. On the other hand, if a process crash before the response value is persistent, i.e., before the operation was completed, then upon recovery the `Recovery` function will complete the operation, together with making the response value persistent. To our knowledge, this is the first definition to consider the affect of a crash on the crashing process, and not only on the object.

One can think of different ways to persist the response value. For example, an operation to a recoverable object gets a location in the shared memory as an extra operand, and the response value is persistent in this location at the response step. This can be implemented easily by replacing Res_p with the supplied location. We do not role out such solutions. However, for ease of presentation the definition uses a simple version.

Notice that any object can be implemented in a recoverable manner, as long as there is no restriction on the correctness condition it requires to satisfy. The `Recover` function does not allow a process p to invoke a new operation before completing the pending operation, hence in every history H a process have at most a single pending operation which is his last invoked operation. Therefore a natural requirement for such an object is linearizability. Since every operation of a process needs to be complete (except for maybe the last one), and we use the original definition of linearizability, this implies that locality holds under this definition.

Definition 5 (Recoverable response linearizability)**.** *A finite history H is recoverable response linearizable if H is recoverable well-formed and after removing all crash and recover steps from H we get a well-formed linearizable history.*

The formal definition allows the use of recoverable objects only, as we require the history to be linearizable under the original Herlihy and Wing's definition. However, such objects can be used in a more general context. One may use recoverable objects only in critical parts of the program, as such an object guarantee the ability to recover and complete the operation in case of a crash. In the rest of the program, assuming data lost is less critical, a simpler objects can be used, e.g., implementations that only guarantee recoverable linearizability. In such a way, the programmer have the flexibility to protect certain parts of the program in the cost of time and space complexity by using the more powerful recoverable objects, while the rest of the program is more efficient but not completely robust to crashes.

3 Recoverable Base Objects

We consider a model in which the program counter (PC) is stored in the non-volatile memory. This can be done either explicitly in the program, or implicitly by the operating system. In this model, upon recovery the last PC is available, and the system knows during what operation the process crash, and thus the proper `Recover` function is invoked. As a result, there is uncertainty regarding whether the last instruction was performed or not. We also assume the `Recover` function has an access to the last PC before the crash, or in other words, it is aware of at what line the crash took place.

The following example clarify why do we need the definition of recoverable object to hold even in the case of primitives. Consider an object supporting a compare-and-swap (CAS) atomic operation. Assume a process p is executing an operation $res \leftarrow C.CAS(old, new)$ followed by a crash. There are several options for at what exact time the crash took place, and each case raises a different problem.

In the standard crash model, a primitive operation is atomic and takes effect instantly in the history, that is, the response follows the invocation in the history, where no other step by any other process is allowed in between. Under the same definition, and assuming the process crash just after completing the CAS operation and before advancing the PC, there was a response step in the history, and thus the RR-linearizability does not require the process to recover the operation. However, the operation is still pending in some sense, as upon recovery the process does not know whether it took affect or not, since the PC still points the same line and res content was erased.

Considering the response of the CAS operation to be at the time where the PC is being advanced solves the former problem. Nevertheless, what if the process crash just after the PC was changed? Again, the operation is not pending, so there is no need to recover it. In this case, upon recovery the process knows the operation was completed, since the PC no longer points to it. However, the process have no access to the response value that was stored in res, residing in the cache, and hence it may not be able to proceed its execution.

A recoverable version of CAS avoids the above problems. The operation considered to be complete only after both the CAS was linearized, and the response value is persistent in Res_p. Therefore, a crash after this point can cause no problem, as the process knows the operation was completed as well as have an access to the response value.

In the following section we present algorithms for implementing recoverable versions for well known primitives. A system equipped with such primitives can be used to implement any object in a recoverable way in the following manner: in case of a crash, the process will simply recover the last primitive operation along which the process crashed. Once the operation complete, the process can continue and execute the remaining code safely. Due to this observation we focus our attention to implementation of recoverable primitives.

The strongest progress property is bounded wait-free, that is, the number of steps a process takes when executing the recovery code in the absence of a failure is finite and bounded by a known constant (may be a function of n, the number of processes in the system), regardless of the other processes steps and the failures the process experienced so far. In addition, we would like the recovery code to use a finite number of variables. Nonetheless, this progress property can not always be achieved, and therefore in some cases we weaken our demand to deadlock-freedom, that is, a process is allowed to wait in the recovery code, and if all processes are taking enough steps then eventually the process will finish its recovery code.

The recoverable object definition requires the response value to be persistent for any operation. However, this requirement is inefficient in case of a trivial response as ack. In this case there is no point to save the response value in the non-volatile memory, as the only information relevant for the process is whether the operation was complete or not. For such an operation we define the response step to be at the point where to PC no longer points the code implementing the operation, and we do not persist the response value. This observation can be used for any operation with a trivial response, even though the recoverable object definition is general and does not handle this case separately.

In addition, the question of having a crash along a `Recover` function needs to be address. One option is to assume a crash can not occur along executing a `Recover` function. In practice it might be reasonable to assume a short time of stability in the system following a crash. However, we wold like to have a stronger model, capable of dealing with crash even along a `Recover` function. In the following implementations, in case of a crash along the `Recover` function the process will simply restart it upon recovery. That is, there is no need for a recover mechanism for the `Recover` function itself.

In the following we use the convention of capital letters names for shared variables and lowercase for local variables. In addition, capital letters are used to refer an operation implementation, and lowercase to refer the respective primitive. A restart response of the `Recover` function implies the operation was not linearized and the process restart it (by moving back to the operation's first line). During the analysis we ignore the ABA problem, as this can be solved easily by augmenting any value with the writing process's id, and a private sequential number. This way we can guarantee no two identical values are used along any execution.

Write For write, we "wrap" the write primitive with a mechanism which allows a process to conclude whether its write or a different write took place since the invocation. For that, process p have a designated variable R_p in the non-volatile memory. The same variable can be used for all write operations of process p.

Algorithm Write program for process p

Shared variables: R_p: composed of two fields, init $< 0, null >$

```
1: procedure WRITE(val)
2:     $temp \leftarrow R$
3:     $R_p \leftarrow < 1, temp >$
4:     $R \leftarrow val$
5:     $R_p \leftarrow < 0, val >$
6:     return $ack$

7: procedure RECOVER()
8:     $< flag, curr > \leftarrow R_p$
9:     if $flag = 0$ and $curr \neq val$ then return restart
10:    if $flag = 1$ and $curr = R$ then return restart
11:    $R_p \leftarrow < 0, val >$
12:    return $ack$
```

Notice that $flag = 0$ whenever p is not performing a WRITE operation, as it is initialise to 0, and before completing a WRITE operation (either in the WRITE code or in the `Recover` function), flag is set back to 0.

The intuition for correctness is the following. A crash before line 3 implies $R_p =< 0, curr >$ where $curr \neq val$, since we assume no two identical values are ever written. In this case the `Recover` function restart the operation. A crush after line 5 implies $R_p =< 1, val >$, and in this case the `Recover` function returns ack. A crash between the two implies $R_p =< 1, curr >$, where $curr \neq val$. If there was a write to R between the two reads of R by p (at the WRITE operation and at the `Recover` function) then either p wrote to R, and we can linearize the WRITE at this point, or there was a write by some other process, and we can linearize the WRITE of p just before it. Hence, the real write "overwrite" p's write, and the rest of the processes can not distinguish between the two scenarios. In any case, we get $curr \neq R$, and the `Recover` function consider the operation as done.

Claim 1. *The WRITE implementation given in Algorithm Write is CRL.*

Proof. First notice the implementation is wait-free and has a constant time complexity for both the WRITE and the `Recover`. Also, the implementation is recoverable - any operation is equipped with a `Recover` function. As discussed earlier in the section, since the response is trivial we weaken our demand and do not require a process to write the response to Res_p.

Consider some execution α. We assume whenever a process crash, once it recovers the system triggers the appropriate `Recover` function if there is a pending operation. Therefore, the corresponding history H describing the execution α is recoverable well-formed, and after removing all crash and recover steps we get a well formed history H'. Following definition 5, it is enough to prove H' is also linearizable.

Since the `Recover` function only reads or write to R_p it does not effect other processes. That is, if a process crash along the `Recover` function then restarting it does not effect other processes. Hence, the proof can ignore crashes along the `Recover` function as long as there is no write to R_p. Assume p performs operation $WRITE(val)$ to variable R in α. If p does not crash along the operation, then obviously p writes to R once in the WRITE operation, and this is also the linearization point.

Assume now there is a crash along the WRITE operation. A crash before line 3 implies p did not wrote neither to R_p nor to R. Upon recovery p reads $R_p =< 0, curr >$ where $curr \neq val$, since we assumed no two identical values are written. Hence the `Recover` function restarts the WRITE code. A crash between the two writes to R_p implies $R_p =< 1, curr >$ where $curr \neq val$. Upon recovery, reading $curr = R$ implies no process wrote to R between the two reads of R, in line 2 and in the `Recover` function. In particular, p did not wrote to R and the WRITE operation is restarted. Otherwise $curr \neq R$, i.e., there was a write to R between the two reads of R. If p wrote to R in the WRITE code, then we can linearize the operation at this point. Otherwise, there is a write by a different process q who took effect. Linearizing p's operation just before q operation, causing q to overwrite the value of p, generates a history which is indistinguishable to all process from one where p does not write at all. In case there are several such operations that needs to be linearize before q's operation, we can linearize them in any order, and the same argument holds. In any case, we can linearize p's operation while respecting the sequential specification, and the `Recover` function returns *ack*. Reading $R_p =< 0, val >$ implies either p crash after line 5, or that the crash was before and there is a `Recover` execution which wrote to R_p. The later happens only when a `Recover` function reads $R_p =< 1, curr >$ and $curr \neq R$, and following the previous analysis, in such a case there is a linearization point for the WRITE operation. □

Compare-and-Swap A Compare-and-Swap (CAS) object supports the $CAS(old, new)$ operation, which atomically sets the value of the object to *new* only if the value it stores is *old*. The operation returns true if the operation succeeded (the write took effect), and false otherwise. CAS object also support a READ operation which returns the value stored in the object.

The main idea is a process first write the value stored in the object before applying the cas primitive. This way, processes informs each other which CAS operations were successful. A process p first read the CAS object. If it observes a value different then *old* it returns false. In such case, the operation is linearized at the read. Otherwise, p informs the last process who committed a successful CAS (before the read) his operation was successful by writing to a designated location in the non-volatile memory. This way, if p crash after a successful CAS operation, the next process to change the value of the CAS, first needs to inform p his CAS was successful. Therefore, upon recovery p can identify its CAS took affect, either because his new value stored in C, or some other process informed him about a success.

Algorithm Compare-and-Swap program for process p

Shared variables:

- C: compare-and-swap object, init $< null, null >$
- R[N][N]: two dimensions array, init $[null, \ldots, null], \ldots, [null, \ldots, null]$

```
procedure CAS(OLD,NEW)
    $< id, val > \leftarrow C.read()$
    if $val \neq old$ then
        $Res_p \leftarrow false$
        return false
    if $id \neq null$ then
        $R[id][p] \leftarrow val$
    $ret \leftarrow C.cas(< id, val >, < p, new >)$
    $Res_p \leftarrow ret$
    return $ret$

procedure RECOVER()
    Read C, R[p][*]
    if $< p, new >$ appears in C, or $new$ appears in R[p][*] then
        $Res_p \leftarrow true$
        return $true$
    else
        return $restart$
```

Following the `Recover` function, if a process crash before applying the cas primitive it will restart its operation, while a recovery from a crash after applying the cas primitive depends on whether the CAS was successful. A key point in the implementation is that a fail CAS operation can be reissued anyway, since it does not effect other processes and thus reissuing it does not violate the sequential specification of the object.

Claim 2. *The recoverable CAS implementation given in Algorithm Compare-and-Swap is CRL.*

Proof. First notice the implementation is wait-free and has a constant time complexity for both CAS and `Recover`. Also, the implementation is recoverable - any operation is equipped with a `Recover` function. Consider some execution α. We assume whenever a process crash, once it recovers the system triggers the appropriate `Recover` function if there is a pending operation. Therefore, the corresponding history H describing the execution α is recoverable well-formed, and after removing all crash and recover steps we get a well formed history H'. Following definition 5, it is enough to prove H' is also linearizable.

Let p be a process performing $CAS(old, new)$ in α. Notice that a CAS value is composed of two fields, an id field stores the last process to perform a successful CAS, and a value field. Assume p does not crash along the CAS operation. First, p reads the content of C. If the value store in C is not *old* then p returns false, and the operation is linearize at the time of the read. Otherwise, it informs the process who last wrote to C his CAS was successful by writing to $R[id][p]$ the value it sees. A two dimensional array is used so that processes will not overwrite each other. Then p tries to change the value of C by performing cas, while considering the id field of the CAS. The operation is linearize at the time of the cas primitive. These are the only two options for linearization points. If there is a linearization point of a successful cas between the read and the cas of p, then since we assumed no two identical values are ever written, this implies the value of C is not *old* at the time when p performs the cas, and indeed the operation returns false. Otherwise, there is no successful cas between the read and cas of p, therefore the value of C at the time of the cas is *old*, and the operation returns true. In any case, this linearization point respects the sequential specification of CAS.

Assume p does crash along a CAS operation. Since the `Recover` function only reads or write to Res_p which is a private variable of p, executing the `Recover` function does not effect other processes. Therefore, if a process crash along the `Recover` function, then restarting it is indistinguishable to other processes from executing it only once. Hence, it is enough to consider the point where p executes the entire `Recover` function with no crash.

If p did not wrote to C, either because the crash was before the cas primitive or a failed cas, then the value *new* of p will never be written to neither C nor $R[p][*]$. This follows from the fact that processes writes only values they have seen in C. As a result, the `Recover` function of p will return restart. Linearizing a failed CAS operation does not effect the other processes, that is, removing this operation is indistinguishable to other processes. Therefore, in both cases, considering the operation as not having a linearization point so far and restarting it does not violate the sequential specification of CAS.

If p did wrote to C, then the next process q to successfully perform cas to C must also write to $R[p][q]$ the value *new*. Since a process performs cas only when it executes the entire CAS operation from start to the cas with no failure (in case of a failure it either restart or return), consider the interval of q performing the CAS operation from the beginning to the successful cas, with no crash. If q read C before the successful cas of p, then the value *val* it reads is different then *new* of p (no two identical values are written). In particular, the value *val* is also different then the value field of C at the time when q performs the cas, contradicting the fact that the cas was successful. Therefore, q read C only after p writes to it, and by our assumption there is no other write to C between the write of p and the read of q (as it the next one to perform a successful cas). It follows that q read $< p, new >$ from C, and thus write $R[p][q] \leftarrow new$ before performing the cas. To conclude, before replacing the value of p stored in C with a new value, it must be that some other process already wrote to $R[p][*]$ the new value of p. Hench, when p executes the `Recover` function, either it will see in C the value it wrote, or that this value has been replaced with a different one, and therefore it will see in $R[p][*]$ the last value it wrote to C. In both cases, p consider the cas operation as successful, and returns true. The linearization point of the operation is at the successful cas primitive.

□

Test-and-Set A Test-and-Set (TAS) object initially stores the value 0, and supports the $T\&S$ operation which atomically change the value of the object to 1 and returns the previous value. A TAS object can be easily implemented using a single CAS object. However, we would like to support an implementation which only uses TAS and read/write primitives in case a process does not crash. It is possible to prove that given only TAS and read/write registers one can not implement a recoverable TAS which is also bounded wait-free, thus the `Recover` function will use waiting mechanism.

We use a doorway mechanism in order to make sure once a process goes through the doorway, all future operations returns 1. This way, we can guarantee the operation who returns 0 can be linearize before any other operation. The doorway can be replaced by reading T, in case it supports also read. A process tries to win the TAS object, and the winning process writes its id into a designated variable $Winner$. Once a process writes to $Winner$ any process can recover by simply read $Winner$. Hence, the main difficulty is to make sure only one process writes to $Winner$, and this is the only process to return 0.

At the beginning of the `Recovery` function, process p first close the *Doorway*, and then announce it is recovering by writing 2 into $R[p]$. In case the crash took effect before the $t\&s$ on T in the $t\&s$ operation, p performs $t\&s$ on T to make sure no process can win the TAS object from this point on. As discussed above, if $Winner$ was written to then p can know if it is the process to win T by comparing its id to the one in T. However, if $Winner$ was not written to, all crashed processes needs to agree on the winner. For that, we use a CAS object for simplicity, although any recoverable leader election algorithm can be used here (for example, one that only uses reads and writes). Notice that there is no need to use recoverable CAS here, since in a crash along the `Recovery` function, the process can simply restart the function.

Once p wins the CAS, it is the only crashed process to win, and therefore it is the candidate to win the TAS object. However, it might be that some other process already won the TAS, but yet to write to $Winner$. To overcome this scenario, p waits for every active process to either finish its t&s operation, or to start the `Recovery` function. We know that any process who will run starting from this point will lose the TAS object, and the CAS object (in case of a crash). Thus, if after the for loop no process wrote to $Winner$ then p can safely announce itself as the winner. Otherwise, p lost to some other active process, and this process wrote to $Winner$ since it won the TAS object in line 3.

Read Following the formal definition of RR-linearizability, a process needs to write the value it reads to Res_p in order to implement a recoverable read. However, such an implementation is redundant and inefficient. In case a process crash before completing its read operation it can simply reissue it upon recovery. In general, operations that only read the status of an object and does not change it (more formally, operation that can be commute with any other operation by a different process), usually uses only read primitive (e.g., snapshot), or at least are implemented in a way such that if a process crash in the middle of an operation then reissue it does not affect the rest of the processes. In such cases, one can implement a recoverable version of the operation by having the `Recover` function restarting the operation.

We now present a way to use these observations in order to implement a recoverable counter in an efficient way. A *counter* object supports an INC operation which atomically adds 1 to the counter value. It also supports a $READ$ operation which returns the counter's value. We first describe a simple implementation which is linearizable under the standard model, and then discuss the changes required for making the implementation also CRL (see Algorithm Counter).

Process p has its own location in an array $R[p]$, which is initialise to 0. For a INC operation, p simply adds 1 to the value of $R[p]$ (by reading and then writing). For $READ$ operation, p simply reads the entire array in a sequential manner, and sum-up the values it see. The correctness argument is as follows. Assume a process p completes a $READ$ operation. Let v be the value it reads from $R[q]$. Since $R[q]$ is monotonically increasing, it must be that $R[q] \leq v$ at the $READ$ invocation and $R[q] \geq v$ at the $READ$ response. Therefore, there are at most v INC operations of q that are linearized before the $READ$ invocation, and at least v INC operations of q that are linearized before the $READ$ response. As this is true for any q, it implies that if p sum up the value val, then there are at most val INC operations that are linearized before the $READ$ invocation, and at least val INC operations that are linearized before the $READ$ response. In particular, there is a point along the $READ$ operation where there are exactly val INC operations that were linearized, and we linearize the $READ$ operation in such a point.

Algorithm Test-and-Set program for process p

Shared variables:

- T: Test-and-Set object, init 0
- C: Compare-and-Swap object, init *null*
- R[N]: an array, init $[0, \ldots, 0]$
- Winner, Doorway: registers, init *null*

```
1: procedure T&S()
2:     if $Doorway \neq null$ then
3:         $Res_p \leftarrow 1$
4:         return 1
5:     $R[p] \leftarrow 1$
6:     $Doorway \leftarrow 1$
7:     $ret \leftarrow T.t\&s()$
8:     if $ret = 0$ then
9:         $Winner \leftarrow p$
10:    $Res_p \leftarrow ret$
11:    $R[p] \leftarrow 2$
12:    return $ret$
```

```
13: procedure RECOVER()
14:     if $R[p] = 0$ then
15:         $restart$
16:     $Doorway \leftarrow 1$
17:     $R[p] \leftarrow 2$
18:     $T.t\&s()$
19:     if $Winner \neq null$ then
20:         $ret \leftarrow (Winner \neq p)$
21:         go to line 31
22:     $C.cas(null, p)$
23:     if $C \neq p$ then
24:         $ret \leftarrow 1$
25:         go to line 31
26:     for $i$ from 0 to $N$ do
27:         $await(R[p] \neq 1)$
28:     if $Winner = null$ then
29:         $Winner \leftarrow p$
30:     $ret \leftarrow (Winner \neq p)$
31:     $Res_p = ret$
32:     return $ret$
```

Algorithm Counter program for process p

Shared variables: R[N]: an array, init $[0, \ldots, 0]$

```
1: procedure INC()
2:     $temp \leftarrow R[p]$
3:     $temp \leftarrow temp + 1$
4:     $R[p].WRITE(temp)$
5:     return $ack$
```

```
6: procedure READ()
7:     $val \leftarrow 0$
8:     for $i$ from 0 to $N$ do
9:         $val \leftarrow val + R[i]$
10:    $Res_p \leftarrow val$
11:    return $val$
```

In order to transform the implementation to a CRL form, we need to both translate the operations to a recoverable version, that is, a process needs to write the response value to Res_p before returning, as well as supplied each operation with a `Recoverable` function. See Algorithm Counter for the code.

The INC operation has a critical point, and this is the write to $R[p]$ which is also the linearization point. Therefore, we need to make sure p writes only once along an INC operation. For that, we replace the atomic write to $R[p]$ with a recoverable version. In case of a crash along the INC operation, if the crash happens before invoking the recoverable $WRITE$, the `Recover` function of the INC operation restart the operation. Otherwise, the `Recover` function of the recoverable $WRITE$ is invoked. For simplicity of presentation we omit the `Recover` function from the code. Notice that the recoverable WRITE requires no two identical values to be written. Although in general this can be achieved by appending any value with the process id and a private sequential number, in our case we can use the original values, as the specification of a COUNTER implies no two identical values are ever written to $R[p]$. As discussed in the beginning of the section, since INC returns a trivial response there is no need to write the response value to Res_p.

For the $READ$ operation, we first add an instruction $Res_p \leftarrow val$ before the return instruction. The `Recover` function of the $READ$ simply restart the $READ$ operation in case there was a crash anywhere along the $READ$ operation.

The above implementation is efficient in terms of the transformation to CRL form. It uses a single recoverable $WRITE$ operations, while the rest of the primitives are not replaced with a recoverable version. Also, the `Recover` functions are almost trivial - either restart or a `Recover` of the $WRITE$ operation. In terms of time and space complexity, the implementation is tight with both for the INC and $READ$ operations, following Jayanti et. al lower bound [13].

4 Discussion

References

[1] M. K. Aguilera and S. Frølund. Strict linearizability and the power of aborting. In *Tech. Rep. HPL-2003-241*, 2003.

[2] R. Berryhill, W. M. Golab, and M. Tripunitara. Robust shared objects for non-volatile main memory. In *19th International Conference on Principles of Distributed Systems, OPODIS*, pages 20:1–20:17, 2015.

[3] K. Censor-Hillel, E. Petrank, and S. Timnat. Help! In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 241–250, 2015.

[4] M. J. Fischer, N. A. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[5] M. Friedman, M. Herlihy, V. J. Marathe, and E. Petrank. Brief announcement: A persistent lock-free queue for non-volatile memory. In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, pages 50:1–50:4, 2017.

[6] W. M. Golab and D. Hendler. Recoverable mutual exclusion in sub-logarithmic time. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC*, pages 211–220, 2017.

[7] W. M. Golab and A. Ramaraju. Recoverable mutual exclusion: [extended abstract]. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC*, pages 65–74, 2016.

[8] R. Guerraoui and R. R. Levy. Robust emulations of shared memory in a crash-recovery model. In *24th International Conference on Distributed Computing Systems (ICDCS)*, pages 400–407, 2004.

[9] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[10] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[11] J. Izraelevitz, H. Mendes, and M. L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Distributed Computing - 30th International Symposium, DISC*, pages 313–327, 2016.

[12] P. Jayanti and A. Joshi. Recoverable FCFS mutual exclusion with wait-free recovery. In *Distributed Computing - 31st International Symposium (DISC)*, 2017.

[13] P. Jayanti, K. Tan, and S. Toueg. Time and space lower bounds for nonblocking implementations. *SIAM J. Comput.*, 30(2):438–456, 2000.

[14] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. Computers*, 46(7):779–782, 1997.

A Formal Proofs

Claim 3. *There exists no CRL recoverable TAS implementation such that a process running without crashing is only allowed to use read, write and t&s primitives, and both TAS and* `Recover` *are wait-free.*

Proof. Let $\mathcal{A}$ be a CRL recoverable TAS implementation such that a process running without crashing is only allowed to use read, write, and t&s primitives. Assume towards a contradiction both TAS and `Recover` are wait-free. Let p and q be two different processes. Notice that solving TAS for two processes is equivalent to consensus, the process who gets 0 as response is the winner. Thus, we use the same proof technique as in the FLP impossibility result [4].

A configuration C is 0-valent (1-valent) if there exists a crash-free extension where p (q) returns 0. C is bivalent if its both 0-valent and 1-valent, and univalent otherwise.

We start with the initial configuration C_0 where both p and q invoke a TAS operation. Then, C_0 is bivalent - a solo run of each process returns 0. Following the FLP proof and since the algorithm is wait-free, there is an extension of C_0 which leads to a bivalent configuration C_1 in which both p and q are about to perform a critical step. This step must be $t\&s$ primitive on the same shared variable. Moreover, a step by any of the processes leads to a different univalent configuration. Denote C_1p to be v-valent, and C_1q to be $\overline{v}$-valent.

Let p and then q perform their next $t\&s$ step, followed by a crash step of p. Since the response of the $t\&s$ primitive can only be stored in the cache of p which is lost due to the crash, upon recovery p does not know whether the $t\&s$ primitive was performed, and what was the response value in case it did. More formally, p can not distinguish between the configurations $C_1 \circ (p, q, CRASH_p)$ and $C_1 \circ (q, p, CRASH_p)$. Therefore, a recover of p after any of these configuration follows by a solo run of p must return the same value ret. Notice that the process must return value since we assumed the `Recover` function is wait-free. This implies both configurations are u-valent for some u, since in a solo run of p followed by a solo run of q one process returns 0.

WLOG assume $u = v$ (the other case is symmetric). Then we consider the configuration $C_1' = C_1 \circ (q, p, CRASH_p)$. Note that C_1' is indistinguishable from a configuration $C_1 \circ (q, p)$ for q, as it does not aware to p's crash. Therefore, C_1' is $\overline{v}$-valent and also v-valent, that is, C_1' is bivalent. By our assumption, q is only allowed to use read, write and t&s primitives, thus we can repeat the same argument proving there is an extension of C_1' leading to a bivalent configuration C_2 where both p and q are about to perform a critical step. Moreover, this step must be t&s primitive on the same shared variable. A simple observation is that this t&s object must be different then any other t&s object that was used, since such an object always returns 1, contradicting the fact both next steps are critical.

The construction continues this way, generating longer and longer executions. We get that there exist executions in which q does not crash, however it access more and more different t&s objects. This contradict the fact the TAS operation is wait-free. □

Claim 4. *The recoverable TAS implementation given in Algorithm Test-and-Set is CRL.*

Proof. We first notice the t&s operation is wait free, and has a constant time complexity. Also, the implementation is recoverable - a process writes the response value to Res_p before returning, and any operation is equipped with a `Recover` function. Consider some execution α. We assume whenever a process crash, once it recovers the system triggers the appropriate `Recover` function if there is a pending operation. Therefore, the corresponding history H describing the execution

α is recoverable well-formed, and after removing all crash and recover steps we get a well formed history H'. Following definition 5, it is enough to prove H' is also linearizable.

If no *t&s* operation was completed in α, then H' is obviously linearizable. Thus, assume there is a complete *t&s* operation, either by completing the *t&s* code, or the `Recover` function. Denote by t the time of the first write to *Doorway*. There must be such a write since we assumed there is a complete operation. Notice that any process who invoke a *t&s* operation after t will return 1 in line 4 - even in case of a crash the `Recover` function restarts the operation, thus if given enough time with no crash eventually it will read 1 from *Doorway* in line 2 and return 1. In addition, there is no operation which returns before t, otherwise there is a process writing to *Doorway* before t, in contradiction.

The proof is composed of two claims - there can be no two operations returning 0, and if all pending operations at time t are completed at least one returns 0. It follows that either there is a single operation active at time t which returns 0 in α, or there is no such operation, but there is an active operation at time t that is yet to complete in α. In both cases we can linearize one operation at time t as returning 0, while the rest can be linearize after it, and they all return 1. This proves H' is linearizable.

Following the code, a process returning 0 implies it wrote to $Winner$ its id. Thus, we prove there can be no two processes writing to $Winner$. Assume there is a process p writing to $Winner$ at the *t&s* code. Hence, p was the first to performs *t&s* on T (as it gets a response of 0). Any other *t&s* on T returns 1, thus there can be no additional process writing to $Winner$ in line 9. Assume there is a process q writing to *winner* in the `Recover` function. Then, q must be the first to perform the *cas* operation in line 22. Any other process who reads C in line 23 gets the value q, thus it will skip to line 31, and does not write to $Winner$. Thus, only q can write to $Winner$ in `Recover` function.

Assume towards a contradiction both p and q writes to $Winner$. Then, p is the first to perform a *t&s* on T in line 7. Notice that p can not crash before the write to $Winner$, otherwise the `Recover` function does not restart the operation, in contradiction to p writing to $Winner$ in the *t&s* code. We get that p writes $R[p] \leftarrow 2$ (either in the *t&s* code, or in the `Recover` function) only after it writes to $Winner$. In addition, q is before line 17 at the time of p performing the *t&s* on T, and in particular before the for loop in the `Recover` function. In order to complete the for loop, q must wait for p to set $R[p] \neq 1$. This in turns happens only after p writes to $Winner$. As a result, whenever q gets to line 28 only after $Winner = p$, hence it does not write to $Winner$ in line 29, in contradiction. Therefore, only one of then can write to $Winner$, and the claim follows.

We now prove there is a process returning 0 if all pending operations at time t have been completed. In fact, we prove in such case there is a process q writing to $Winner$. Since q is the only process to write to $Winner$ and it complete its operation, it must return 0, either in the *t&s* code or at the `Recover` function. First notice that any process who sets $R[p] \leftarrow 1$ is active at time t, thus the operation is complete in α. Also, there exist such a process, the process who writes to *Doorway* for example.

Assume toward a contradiction no process writes to $Winner$. If no process crash after setting $R[p] = 1$ then the first process to perform the *t&s* on T in line 7, also complete its operation and writes to $Winner$ in line 9, in contradiction. Thus, there exists a process who crash and complete the `Recover` function after setting $R[p] \leftarrow 1$. By our assumption $Winner = null$, hence the if in line 19 returns false. Let q be the first process to perform the *cas* operation in line 22. q sets the value of C to be q, and this does not change along the execution (any other *cas* fails). We get that starting from this point $R[q] = 2$, $Winner = null$ and $C = q$. Hence, even if q crash and restart the `Recover` function many times, eventually it complete its operation, thus it does not use any of the go to instructions, and after completing the for loop it reads $null$ from $Winner$. Therefore, in line 29 q writes to $Winner$, in contradiction.

$\square$