Recoverable and Detectable Self-Implementations of Swap

Tomer Lev Lehman
Department of Computer Science, Ben Gurion University, Beer Sheva, Israel

Hagit Attiya
Department of Computer Science, Technion, Haifa, Israel

Danny Hendler
Department of Computer Science, Ben Gurion University, Beer Sheva, Israel

Abstract

Recoverable algorithms tolerate failures and recoveries of processes by using non-volatile memory. Of particular interest are *self-implementations* of key operations, in which a recoverable operation is implemented from its non-recoverable counterpart (in addition to reads and writes).

This paper presents two self-implementations of the *swap* operation. One works in the *system-wide failures* model, where all processes fail and recover together, and the other in the *independent failures* model, where each process crashes and recovers independently of the other processes.

Both algorithms are wait-free in crash-free executions, but their recovery code is blocking. We prove that this is inherent for the independent failures model. The impossibility result is proved for implementations of *distinguishable* operations using *interfering* functions, and in particular, it applies to a recoverable self-implementation of swap.

2012 ACM Subject Classification Theory of computation → Shared memory algorithms

Keywords and phrases Multi-core algorithms, persistent memory, non-volatile memory, recoverable objects, detectablitly

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.24

Related Version *Full Version*: https://arxiv.org/abs/2308.03485

Funding Partially supported by the Israel Science Foundation (grant 22/1425).

1 Introduction

Recent years have seen a rising interest in the failure-recovery model for concurrent computing. This model captures an unstable system, where processes may crash and recover. Two variants of the model have been considered. In the *system-wide failures* model (also called the *global-crash* model), all processes fail simultaneously and a single process is responsible for the recovery of the whole system. In the *independent failures* model (also called the *individual-crash* model), each process can incur a crash independently of other processes and recovers independently. *Recoverable* algorithms, tolerating failures and recoveries, have been presented for various concurrent data structures, for both the system-wide failures model [9, 11, 14, 17, 24, 34, 37, 40] and the independent failures model [2, 4, 9, 34, 37].

The correctness of a recoverable algorithm can be specified in several ways. *Durable Linearizability* [26] intuitively requires linearizability [23] of all operations that survive the crashes. *Detectability* [17] ensures that upon recovery, it is possible to infer whether the failed operation took effect or not and, in the former case, obtain its response. *Nesting-safe Recoverable Linearizability* (NRL) [2], originally defined for the independent failures model, ensures detectability and linearizability.

NRL is defined in a way that facilitates the nesting of recoverable objects. By providing implementations of NRL primitive operations on base objects, a programmer can combine several of these primitives to create recoverable implementations of complex higher-level objects and algorithms. This level of abstraction can be helpful in the adoption of recoverable algorithms.

To facilitate high-level implementations of complex NRL objects, it is helpful to introduce implementations of low-level NRL objects. An attractive approach to designing low-level NRL objects is through *self-implementations* [37], in which a recoverable operation is implemented with instances of the *same* primitive operation, possibly with additional reads and writes on shared variables. This approach ensures that when using the recoverable version of an operation, the system must only support its hardware-implemented primitive counterpart.

NRL self-implementations already exist for various primitives, including *read*, *write*, *test&set*, and *compare&swap* [2, 4, 27], as well as *fetch&add* [37]. A universal construction using NRL read, write and *compare&swap* objects [4] builds upon previously-introduced self-implementations of NRL objects to take any concurrent program with read write and CAS, and make it recoverable while adding only constant computational overhead.

This paper presents the first NRL self-implementations of *swap*, for both the system-wide and the independent failures models. Swap is a widely-used primitive that is employed by concurrent algorithms. Our implementations borrow ideas from the *recoverable mutual exclusion (RME)* [20] algorithms of [19, 29], which use a similar approach to overcome swap failures. Unlike these algorithms, however, our implementations are also challenged with the task of satisfying wait-freedom and linearizability. Both our algorithms are wait-free in crash-free executions, while the recovery code in both is blocking.

We also prove the impossibility of implementing a class of *distinguishable operations* using a set of *interfering functions* [22] in a recoverable *lock-free* fashion in the independent failures model. In particular, this result applies to self-implementations of swap, but it also holds for, e.g., implementing swap using a combination of fetch-and-add and swap. Other distinguishable operations to which this proof applies are the *deque* of a queue object and the *pop* of a stack object. Our impossibility result unifies and extends specialized results for self-implementations of *test&set* [2] and *fetch&add* [37]. Another related impossibility result addresses recoverable consensus in the independent failures model [18].

To summarize, our contributions are the following:

- A recoverable detectable self-implementation of swap in the system-wide failures model.
- A recoverable detectable self-implementation of swap in the independent failures model.
- An impossibility proof for implementations of distinguishable operations using interfering functions in the independent failures model.

Related Work

Several correctness conditions and definitions, in addition to those previously mentioned, exist for recoverable algorithms. *Strict linearizability* [1] requires operations interrupted by a failure to take effect either before the failure or not at all. A relaxed condition called *persistent atomicity* [21], defined for message-passing systems, allows an operation to take effect even after a failure, before the next operation invocation of the same process. *Recoverable linearizability* [6] builds on the definition of persistent atomicity, but it also ensures *locality* – the desirable property that an execution involving multiple objects is correct if and only if its projection onto each individual object is correct [23].

Ben-David, Friedman and Wei [5] define a hierarchy based on the sets of execution histories that are allowed by each of the above definitions under the individual-crash model, in which the same processes are allowed to be invoked following a crash. In this hierarchy,

NRL is not implied by the above definitions, but it does overlap with some of them. In particular, any strictly linearizable or persistent atomic history completion in which no operation was removed is also consistent with NRL. Since NRL only requires the completion of an operation before the next one by the same process, a history may be NRL consistent even if some process executes an entire operation after the crash before another process completes its operation which was started before the crash. Such a history is not strictly linearizable. However, it is still persistently atomic. In fact, persistent atomicity is strictly stronger than NRL under this history completions interpretation.

Detectable sequence specifications (DSS) [34] also formalizes the notion of detectability. DSS is more portable and less reliant on system assumptions in comparison to NRL, but it delegates the responsibility for nesting to the application code.

Several papers introduce general mechanisms to port existing algorithms and make them persistent, e.g., by using transactional memory [7,11,25,39], universal constructions [6,10,12], or for specific families of algorithms [4, 13, 16]. Most of these transformations use strong primitives such as *CAS* while their non-recoverable counterparts may use weaker primitives, in terms of their consensus number [22]. We believe future research may use our self-implementation of swap to extend general constructions such as [4], mentioned above, to programs that also use swap as a primitive. In addition, using NRL self-implementations can ensure the ability to implement higher-level objects using the same primitives as would be used in a non-recoverable setting. In contrast, other works [9, 14, 17, 34, 35, 40] mostly rely on strong primitives such as *CAS*.

Other papers present hand-crafted persistent implementations of specific data structures, e.g., [17,38,41,42]. In contrast to these implementations, our algorithms provide a recoverable counterpart to an atomic primitive operation, which we believe can later be used in various other implementations of recoverable algorithms.

Our algorithm for the independent failures model uses an RME lock such as the one presented by Golab and Ramaraju [20], which uses only reads and writes. Additionally, there is a long line of research on RME, solving several other aspects such as abortability, FCFS, and more (see, e.g., [8, 28, 30–33]).

2 Model and Definitions

Since this work deals with a recoverable version of a primitive object, we assume a simplified version of the NRL system model [2], which does not capture nesting. There are n asynchronous *processes* $p_1, \ldots, p_n$, which communicate by applying atomic *primitive* read, write and read-modify-write operations to *base objects*. The state of each process consists of non-volatile *shared variables*, which serve as base objects, as well as volatile *local variables*.

We first describe the *independent failures* model, in which each process can incur a *crash-failure* (or simply a *crash*) at any point during the execution independently of other processes. A crash resets all of its local variables to arbitrary values but preserves the values of all non-volatile variables.

A process p *invokes an operation Op* on an object by performing an *invocation step*. *Op completes* by executing a *response step*, in which *the response of OP is stored to a local volatile variable of p*. It follows that the response value is lost if p incurs a crash before *persisting* it, that is, before writing it to a non-volatile variable. Operation *Op* is *pending* if it was invoked but was not yet completed; a process has at most one pending operation.

Each *recoverable operation Op* has an associated *recovery procedure Op*.RECOVER, which is responsible for completing *Op* upon recovery from a crash. If the object supports a *single* recoverable operation, then its recovery procedure is simply named RECOVER. Following

a crash of process p that occurs when p has a pending recoverable operation instance, the system eventually resurrects process p by invoking the recovery procedure of the recoverable operation that was pending when p failed. This is represented by a *recovery step for p*.

Formally, a *history* H is a sequence of *steps*. There are four types of steps:

1. An *invocation step*, denoted (INV, p, O, Op), represents the invocation by process p of operation Op on object O.
2. A *response step* s, denoted (RES, p, O, Op, ret), represents the completion by process p of operation Op invoked on object O by some (invocation) step s' of p, with response ret being written to a local variable of p; *s is the response step that matches s'*. An operation Op can be completed either normally or when, following one or more crashes, the execution of Op.RECOVER is completed.
3. A *crash step* s, denoted $(CRASH, p)$, represents the crash of process p. We call the recoverable operation Op of p that was pending when the crash occurred the *crashed operation of s*. $(CRASH, p)$ may also occur when the recovery procedure Op.RECOVER is executed for recovering an operation of p and we say that Op is the crashed operation of s also in this case.
4. A *recovery step s for process p*, denoted (REC, p), is the only step by p that is allowed to follow a $(CRASH, p)$ step s'. It represents the resurrection of p by the system, in which it invokes Op.RECOVER, where Op is the crashed operation of s'. We say that *s is the recovery step that matches s'*.

An object is *recoverable* if all its operations are recoverable. Below, we consider only histories that arise from operations on recoverable objects or atomic primitive operations.

When a recovery procedure Op.RECOVER is invoked to recover from a crash represented by step s, we assume it receives the same arguments as those with which Op was invoked when that crash occurred.

It was shown [3] that detectable algorithms for the NRL model must keep an auxiliary state that is provided from outside the operation, either via operation arguments or via a non-volatile variable accessible by it. We assume that Op.RECOVER has access to a designated per-process non-volatile variable SEQ_p, storing *the sequence number of Op*. Before p invokes an operation on the object, it increments SEQ_p.

Consider a scenario in which p incurs a crash (represented by a crash step s) immediately after a recoverable operation Op completes (either directly or through the completion of Op.RECOVER); this event is represented by a response step r. In this case, the response value is lost and, moreover, Op.RECOVER will not be invoked by the system, since the crashed operation of s is no longer Op. In general, p may therefore not be able to proceed correctly. Hence, it is sometimes required to guarantee that a recoverable operation returns only once its response value gets persisted. This is defined formally as follows.

▶ **Definition 1.** *A recoverable operation Op is* strictly recoverable *[2] if whenever it is completed, either directly or by the completion of Op.Recover, with a (RES, p, O, Op, ret) step event, ret is stored in a designated persistent variable accessed only by process p.*

A history H is *crash-free* if it contains no crash steps (hence also no recovery steps). $H|p$ denotes the sub-history of H consisting of all the steps by process p in H. $H|O$ denotes the sub-history of H consisting of all the invocation and response steps on object O in H, as well as any crash step in H, by any process p, whose crashed operation is an operation on O and the corresponding recovery step by p (if it appears in H). $H|<p, O>$ denotes the sub-history consisting of all the steps on O by p.

A crash-free sub-history $H|O$ is *well-formed*, if for all processes p, $H|<p, O>$ is a sequence of alternating, matching invocation and response steps, starting with an invocation step. A crash-free history H is *well-formed* if:

1. $H|O$ is well-formed for all objects O, and
2. each invocation event in $H|p$, except possibly the last one, is immediately followed by its matching response step.

$H|O$ is a *sequential object history* if it is an alternating series of invocations and the matching responses starting with an invocation; it may end with a pending invocation. The *sequential specification* of an object O is the set of all *legal* sequential histories over O. H is a *sequential history* if $H|O$ is a sequential object history for all objects O.

Two histories H and H' are *equivalent*, if $H|<p, O> = H'|<p, O>$ for all processes p and objects O. Given a history H, a *completion* of H is a history H' constructed from H by selecting separately, for each object O that appears in H, a subset of the operations pending on O in H and appending matching responses to all these operations, and then removing all remaining pending operations on O (if any).

An operation op_1 precedes an operation op_2 in H, denoted $op_1 <_H op_2$, if op_1's response step is before the invocation step of op_2 in H.

▶ **Definition 2** (Linearizability [23], rephrased)**.** *A finite crash-free history H is* linearizable *if it has a completion H' and a legal sequential history S such that H' is equivalent to S and $<_H \subseteq <_S$ (i.e., if $op_1 <_H op_2$ and both op_1 and op_2 appear in S, then $op_1 <_S op_2$).*

To define nesting-safe recoverable linearizability, we introduce a more general notion of well-formedness that also applies to histories that contain crash/recovery steps. For a history H, we let $N(H)$ denote the history obtained from H by removing all crash and recovery steps. A history H is *recoverable well-formed* if every crash step in $H|p$ is either p's last step in H or is followed in $H|p$ by a matching recovery step of p, and $N(H)$ is well-formed.

▶ **Definition 3** (Nesting-safe Recoverable Linearizability (NRL))**.** *A finite history H satisfies* nesting-safe recoverable linearizability *(NRL) if it is recoverable well-formed and $N(H)$ is a linearizable history. An object implementation satisfies NRL if all of its finite histories satisfy NRL.*

We build upon the above definitions also for the *system-wide failures* model, but we require that if a crash step occurs, then it occurs simultaneously for all processes whose operations crash. Formally, let $p_{i_1}, \ldots, p_{i_k}$, for $k \leq n$, be the set of processes that have a pending invocation of operation Op when a system-wide crash occurs. The crash is represented by appending the sequence $(CRASH, p_{i_1}), \ldots, (CRASH, p_{i_k})$ to the execution. During recovery, the system executes a parameterless *global recovery procedure for Op* called Op.GRECOVER, which is represented by appending the sequence $(REC, p_{i_1}), \ldots, (REC, p_{i_k})$ to the execution. Once Op.GRECOVER completes, the system resurrects each of the processes $p_{i_1}, \ldots, p_{i_k}$ to perform an *individual recovery procedure for Op*, called Op.RECOVER. New operations on the object can be invoked only after recovery ends. If Op is a single object operation, we write GRECOVER and RECOVER instead of Op.GRECOVER and Op.RECOVER, respectively.

An algorithm is *lock-free* if, whenever a set of processes take a sufficient number of steps and none of them crashes, then it is guaranteed that one of them will complete its operation. An algorithm is *wait-free*, if any process that does not incur a crash and takes a sufficient number of steps completes its operation in a finite number of its steps. A swap object supports the *swap(val)* operation, which atomically swaps the object's current value *cur* to *val* and returns *cur*. For presentation clarity, we denote from now on the primitive swap operation by (lowercase) swap and its counterpart implemented recoverable operation by (uppercase) SWAP.

3 Detectable Swap Algorithm for the System-Wide Failures Model

A key challenge to overcome when implementing a detectable swap object from read, write, and primitive swap operations is that the return values of one or more primitive swap operations may be lost upon a system-wide failure that occurs before they are persisted. These operations may have already affected the state of the swap object and, moreover, operations by other processes may have already returned the values written by these primitive operations. To ensure linearizability, the implementation must identify such operations and handle them correctly.

The return value of each SWAP operation must meet a few requirements. First, it should be the input of another SWAP operation (or the initial value of the swap object). Second, the operand swapped in by one SWAP operation can be returned by at most a single other SWAP operation. Finally, in order to maintain linearizability, if $op_1 <_H op_2$ holds, then op_1 cannot return the value that was swapped in by op_2.

Figure 1 illustrates a scenario involving six processes, $p_1, \ldots p_6$, performing eight SWAP operations, $op_0, \ldots op_7$. A system-wide crash occurs when operations op_0, op_2, op_4, op_6 have already been completed (their return values are specified in Figure 1) while operations op_1, op_4, op_5, op_7 are pending. Note that operations op_1, op_4, op_5, although not completed, have affected the global state of the swap object as their inputs are the return values of other operations, while op_7 (pending as well) might or might not have affected the object's state.

There are several ways the system may recover in order to produce a correct linearizable result. In all of them, op_1 must return 0. The remaining uncompleted operations might return different values in the following ways:

1. op_4 returns 2, op_5 returns 3, and op_7 returns 6.
2. op_7 returns 2, op_4 returns 7, and op_5 returns 3.
3. op_4 returns 2, op_7 returns 3, and op_5 returns 7.

In this example there are several possible linearizations because op_7 may be linearized in several ways since its effect on the global state is unknown. Note that for correctly recovering op_1, the operand of the very first operation, op_0, must be recorded. This is why our algorithms retain a record of all invoked operations.

We represent the order of SWAP operations as a linked list of Node structures, the end of which is pointed by a *tail* variable manipulated using primitive swaps. The list starts with a sentinel node called *headNode*, which holds the object's initial value (denoted $\perp$).

Each Node structure represents a single SWAP operation and stores a pointer *prev* to the node of its predecessor operation and the SWAP's operand *val*. The order of SWAP operations is reflected by the order of the Node structures in the list. By doing so, each Node points to the previous Node structure that represents the previous SWAP operation, hence the operation's return value will be *Node.prev.val*.

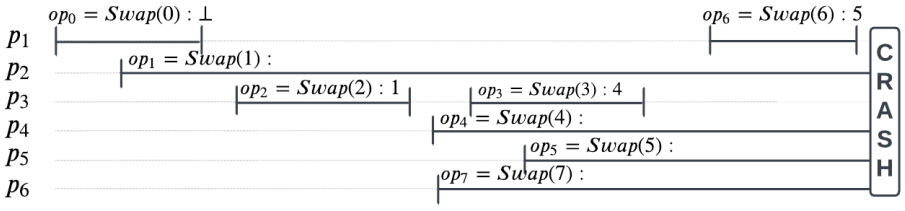

Figure 1 An example of the effect of a system-wide failure.

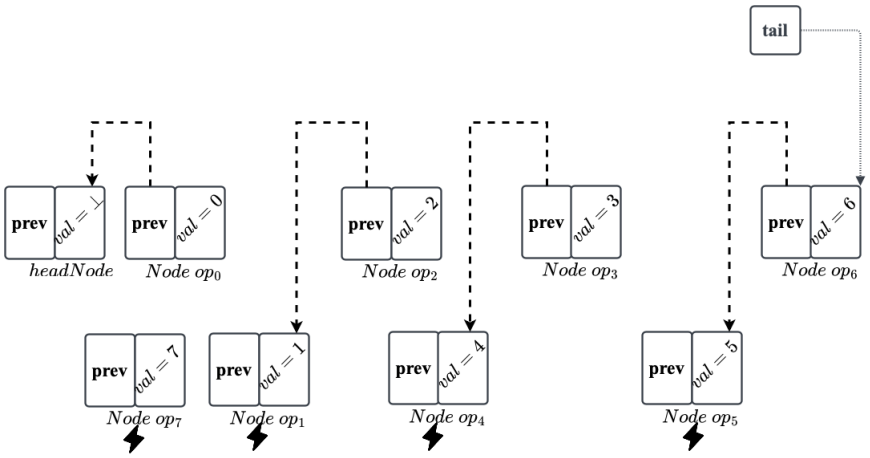

Figure 2 The fragments corresponding to Figure 1. Operations 1,4,5 crashed after swapping their Nodes to *tail* but before persisting their pointer to their predecessor Node.

A problem can occur if a process successfully swaps its Node into the list but crashes before pointing from its structure to the previous Node. This type of failure may create what is referred to as *fragments* in the list representing the SWAP operations. Thus, instead of a single complete list, crashes may result in several incomplete disconnected lists. In order to reconnect these fragments back to a complete list, our algorithms go over all previously-announced operations upon recovery and recreate a correctly ordered complete list of operations.

A similar idea was used by the recoverable mutual exclusion (RME) algorithms of Golab and Hendler [19] and Jayanti, Jayanti and Joshi [29]. These algorithms also have to reconnect the fragments of an MCS lock [36] linked-list based queue, caused by failures that occur just before or after primitive swap operations. However, our algorithms need to address two challenges that do not exist in RME algorithms. First, the SWAP operations of our algorithms are required to be wait-free whereas RME implementations are allowed to block. Second, unlike RME implementations, our algorithms are required to maintain linearizability. Specifically, the new order of list fragments, constructed during recovery, must respect the real-time order between SWAP operations.

We address these challenges by employing a *fragment ordering* scheme, which we view as the key novelty of our algorithms. The scheme encapsulates the critical steps of each SWAP operation by two vector timestamp computations. Based on the resulting timestamps, the recovery code ensures the following invariant: if a fragment A contains a Node n_A that was created after an operation associated with a Node n_B on fragment B was completed, then fragment B will be ordered after fragment A in the linked list. We formally define this fragment ordering scheme in Definition 4.

Figure 2 presents the fragments corresponding to Figure 1. We describe it next and introduce a few terms that are used later. The fragment of op_7 contains a single Node; we name such 1-size fragments *singleNodes*. The node of op_0 belongs to the single fragment that contains *headNode*; we name this fragment the *head fragment*. The nodes of op_5 and op_6 belong to the single fragment that contains the node pointed to by *tail*; we name this fragment the *tail fragment*. Fragments such as (op_2, op_1) and (op_3, op_4) that are neither singleNodes, nor head nor tail fragments are called *middle fragments*.

Algorithm 1 Recoverable detectable SWAP for system-wide failures model, code for process i; text in blue is for the independent failures algorithm only.

Define Node: struct {*val* : **Value**, *prev* : **ref** to Node, *prevExecution* : **ref** to Node, *startVts* : **vector**, *endVts* : **vector**, *seq* : **int**, *inWork* : **int**}

Initial State:

$Nodes[i] \leftarrow null$ for $i \in \{0, \ldots, n\}$
$VTS[i] \leftarrow 0$ for $i \in \{1, \ldots, n\}$
$headNode \leftarrow new(Node)$
$headNode.val \leftarrow \perp$, $headNode.seq \leftarrow 0$, $headNode.prev \leftarrow null$, $headNode.prevExecution \leftarrow null$, $headNode.inWork \leftarrow 0$.
$headNode.startVts \leftarrow collect(VTS)$, $headNode.endVts \leftarrow collect(VTS)$
$tail \leftarrow headNode$
$Nodes[0] \leftarrow headNode$

1: **procedure** SWAP(*val*) ▷ executed by process i
2: $myNode \leftarrow new(Node)$
3: $myNode.prev \leftarrow$null, $myNode.seq \leftarrow SEQ_i$, $myNode.prevExecution \leftarrow$ null, $myNode.val \leftarrow$val
4: $VTS[i] \leftarrow VTS[i] + 1$
5: $myNode.startVts \leftarrow collect(VTS)$
6: $prevExecution \leftarrow Nodes[i]$
7: $myNode.prevExecution \leftarrow prevExecution$
8: $myNode.inWork \leftarrow 1$ ▷ begin swap
9: $Nodes[i] \leftarrow myNode$ ▷ announce the operation
10: $prev \leftarrow swap(\&tail, myNode)$
11: $myNode.prev \leftarrow prev$ ▷ persisting operation
12: $myNode.endVts \leftarrow collect(VTS)$
13: $myNode.inWork \leftarrow 0$ ▷ finished operation
14: $Res_i \leftarrow myNode.prev.val$
15: **return** Res_i

When ordering middle fragments and singleNodes, the algorithm uses vector timestamps for maintaining linearizability. As an example, consider a linked list, reconnecting the fragments of Figure 2, in which $op_4.prev \leftarrow op_0$, $op_1.prev \leftarrow op_3$, $op_7.prev \leftarrow op_2$ and $op_5.prev \leftarrow op_7$. Although this list contains the Nodes of all operations from tail to head, it violates linearizability because op_3 is ordered after op_2 although it follows it in real-time order. By using the two vector timestamps, our algorithm is able to order the fragments so that linearizability is maintained.

Another case that may arise is a set of fragments that consists of a single complete list with one or more singleNodes, which results from a crash that occurs when none of the pending operations completed their primitive swap operations. As we prove, in this case it is safe to put all these singleNodes (in any order) at the end of the list.

3.1 Detailed Description of the Algorithm

Data structure definitions and the pseudo-code are presented by Algorithm 1. Text in blue is for the independent failures algorithm and should be disregarded for now. We first describe key data structures and shared variables.

Each SWAP operation is represented by a single *Node* structure. *Node.val* stores the operand of the SWAP operation. *Node.seq* stores the sequence number of the current process's SWAP operation. *Node.prev* is a pointer to the *Node* structure representing the previous SWAP operation. Consequently, *Node.prev.val* stores the value that must be returned by the SWAP operation represented by *Node*. Each *Node* structure also stores two timestamp

16: **procedure** GRECOVER() ▷ Global SWAP recovery procedure
17: $V \leftarrow \emptyset$, $E \leftarrow \emptyset$
18: **for** i from 0 to n **do**
19: $currNode \leftarrow Nodes[i]$
20: **while** $currNode \neq null$ **do**
21: $V \leftarrow V \cup \{currNode\}$
22: **if** $currNode.prev \neq null$ **then**
23: $V \leftarrow V \cup \{currNode.prev\}$
24: $E \leftarrow E \cup \{(currNode, currNode.prev)\}$
25: $currNode \leftarrow currNode.prevExecution$
26: $TAILNODE \leftarrow new(Node)$
27: ▷ Add graph node representing the list's tail and graph edge pointing to the *tail* Node
28: $V \leftarrow V \cup \{TAILNODE\}$
29: $E \leftarrow E \cup \{(TAILNODE, tail)\}$
30: Compute set $Paths$ of maximal paths in graph $\mathcal{G} = (V, E)$
31: $MiddlePaths \leftarrow \emptyset$
32: $SingleNodes \leftarrow \emptyset$
33: **for** $path \in Paths$ **do**
34: **if** $len(path) == 1$ **then**
35: $SingleNodes \leftarrow SingleNodes \cup \{start(path)\}$
36: Remove $path$ from $Paths$
37: **for** $path \in Paths$ **do**
38: **if** $TAILNODE \in path$ and $Nodes[0] \in path$ **then**
▷ There is a single full path from tail to head
39: For every Node in $SingleNodes$ re-execute SWAP from Line 10
40: **return**
41: **else if** $TAILNODE \in path$ **then**
42: $TailPath \leftarrow path$
43: **else if** $Nodes[0] \in path$ **then**
44: $HeadPath \leftarrow path$
45: **else**
46: $MiddlePaths \leftarrow MiddlePaths \cup \{path\}$
47: **for** $curPath \in sort(MiddlePaths \cup SingleNodes$ in non-increasing $\succ$ order) **do**
48: $end(TailPath).prev \leftarrow start(curPath)$
49: update $TailPath$ to include added path
50: $end(TailPath).prev \leftarrow start(HeadPath)$
51: **procedure** RECOVER(val) ▷ Individual SWAP recovery procedure for process i
52: $myNode \leftarrow Nodes[i]$
53: **if** $myNode == null$ or $myNode.seq < SEQ_i$ **then**
54: **return** SWAP(val)
55: **else**
56: $Res_i \leftarrow myNode.prev.val$
57: **return** Res_i

vectors of size n, *Node.startVts* and *Node.endVts*. Lastly, *Node.prevExecution* is a pointer to the *Node* structure representing the previous SWAP operation by the same process (if there is one).

Nodes is an array of $n + 1$ pointers to *Node* structures. *Nodes*[0] points to the *headNode* sentinel node. For each process $i \in \{1, \ldots, n\}$, *Nodes*[i] points to the beginning of a list of *Node* structures, induced by *prevExecution* pointers, which represent the SWAP operations of process i. This array is used for recording all *Node* structures created throughout the execution.

VTS is an array of length n that serves as a global vector timestamp. Entry i counts the number of operations performed by process i. *tail* is a pointer to a *Node* structure. Res_i is a per-process non-volatile variable to which an operation's response is persisted before it returns. The algorithm maintains a linked list of *Node* structures representing the order of SWAP operations and tail points to the last *Node* structure in the list.

The following order relation between fragments is used by the global recovery procedure.

▶ **Definition 4.** *Given two fragments A and B, we denote $A \succ B$ if there are nodes $n_A \in A$ and $n_B \in B$ such that $n_A.startVts > n_B.endVts$. If neither $A \succ B$ nor $B \succ A$ holds, we say that A and B are $\succ$-equal.*

A SWAP operation first creates a *Node* structure and initializes it (Lines 2–3). It then increments its entry of the VTS, collects VTS, and writes the resulting vector timestamp to the *startVTS* field of its node (Lines 4–5). In Lines 6–7, the node representing the current operation is linked to the list of the previous operations executed by this process. Then, the process *announces the operation* by writing a pointer to its node to its entry of the *Nodes* array (Line 9). Next, the procedure invokes an atomic swap operation to read a node pointer from *tail* and swap it with a pointer to the node representing the current operation (Line 10). Then, the previous *tail* value is persisted to the *prev* field of the operation's *Node* structure (Line 11), thus adding this operation to the fragment of its predecessor operation. If a process executes Line 10 but the system crashes before it executes Line 11, a new fragment will result that cannot be reached from *tail* using *prev* pointers. These fragments are reconnected by the global recovery procedure GRECOVER. The operation terminates by performing a second collect of VTS, writing it to the *endVts* field, and returning the previous value stored at *prev.val* after writing it to its *Res* variable (Lines 12, 14–15).

The GRECOVER procedure (Lines 16–50) of the SWAP operation is performed by the system upon recovery. As we've mentioned before, its task is to reconnect fragments caused by a system-wide crash by creating a total order between SWAP operations that maintains linearizability. When it terminates, all the SWAP operations that were previously announced (in Line 9) are ordered in a single fragment that includes the *headNode* and the node pointed to by *tail*. As we prove, the order of operations induced by this fragment is a linearization of the execution.

After the completion of GRECOVER, the system resurrects all the processes whose SWAP operations crashed, for executing the individual recovery procedure (Lines 51–57). It first checks if the sequence number of the process' last announced operation (found in the *Nodes* array) equals SEQ_i. If it does, the value of *Nodes*[i]'s *prev* field is written to Res_i and returned (Lines 56–57). Otherwise, process i's operation crashed before it was announced and so SWAP is re-executed (Line 54).

Pending SWAP operations that updated their *prev* pointers before the crash can simply return the value stored in *prev.val*. SWAP operations that did not update their *prev* pointer in Line 11 before a crash are of two types: Those that executed Line 10 before the crash and those that did not. SWAP operations of the latter type are simpler to deal with since they did not change the *tail* pointer and can therefore be re-executed. Correctly ordering SWAP operations of the first type (i.e. those that executed Line 10 but did not execute Line 11) is more challenging, since their primitive swap operation changed *tail*'s value but its return value was lost.

We proceed to describe the GRECOVER procedure. It starts by constructing a directed graph $\mathcal{G}$ whose nodes correspond to Node structures and whose edges correspond to *prev* pointers (Lines 17–29). The construction is done by traversing (non-null) *prev* and *prevExecution* fields starting from each entry of the Nodes array (Lines 17–25). After the traversal

ends, a special TAILNODE node and an edge directed from it to the node pointed at by *tail* are added to the graph for simplifying the handling of the tail fragment (Lines 26–29). A set *Paths* of maximal directed paths in $\mathcal{G}$ is computed in Line 30. $\mathcal{G}$ is cycle-free, because each SWAP operation performs Line 10 at most once and, if it does, receives in response a pointer to an operation that performed Line 10 before it. Thus, the set *Paths* is well-defined. Each element of *Paths* represents a fragment.

Next, all singleNodes (if any) are removed from *Paths* and inserted into a separate *SingleNodes* set (Lines 32–36). If *Paths* has a fragment that contains both TAILNODE and the *headNode* sentinel node, then, as we prove, there are no middle fragments. In this case, each of the operations that correspond to *SingleNodes* is executed, in turn, starting from Line 10 and their nodes are thus appended to the end of this full path. Then GRECOVER returns (Lines 38–40). Otherwise, the fragments in *Paths* are categorized to a single *HeadPath* fragment, a single *TailPath* fragment, and a *MiddlePaths* set that contains all other paths, which are middle fragments (Lines 37–46).

Next, all fragments other than *HeadPath* and *TailPath* are sorted in non-increasing $\succ$ order (see Definition 4) and are appended, one after the other, to the end of *TailPath* by updating appropriate *prev* fields (Lines 47–49). Sorting is done by comparing *startVts* and *endVts* fields as specified by Definition 4. The construction of the full order is concluded by appending the *HeadPath* to the end of the *TailPath* (Line 50).

The execution of GRECOVER, as well as that of the individual recovery procedure, can incur one or more crashes. In this case, the recovery process is re-executed upon recovery from each crash. As we prove, when it completes, operations are ordered correctly.

3.2 Proof of Correctness

We say that the list $l = (a, a_1 = a.prev, a_2 = a_1.prev..., a_m = a_{m-1}.prev, b = a_m.prev)$, for $m \geq 0$, of length $m + 2$, is *induced by prev pointers*. We also say that l *starts at* a and l *ends at* b. We define the notion of a list *induced by prevExecution pointers* similarly.

The following lemma states that in GRECOVER at Line 30, all Node structures ever announced by Line 9 are in V, and all *prev* pointers from Node u to Node v that were set in Line 11 are represented by an edge $(u, v) \in E$.

▶ **Lemma 5.** *At Line 30,* $\{v | v$ *was announced in Line 9}* $\subseteq V \wedge \{(u, v) | u.prev = v\} \subseteq E$ *holds.*

Proof. First note that every Node structure announced at Line 9 is inserted into the Nodes array. Also, notice that each entry in the Nodes array is written to by a single process.

In addition, every time a Node x is overwritten by a new Node y in Nodes[p] by process p, $y.prevExecution$ equals x. It follows that all of the Node structures created by p can be reached by going over the list induced by *prevExecution* pointers starting from $Nodes[p]$, for all $p \in \{1 \dots n\}$. Also notice that Nodes[0] is a special case of a list of length 1, representing the initial state.

In the for loop at Line 18, the recovery process goes over all process numbers from 1 to n and also over Nodes[0]. For each process, it goes over the list induced by its *prevExecution* pointers, so that *currNode* gets assigned every Node ever announced in Line 9 as well as Nodes[0]. Consequently, $\{v | v$ was announced in Line 9$\} \subseteq V$ holds in Line 30. In addition, by going over every announced Node and adding the $TAILNODE$ link, we ensure that for every pair of Node structures u and v, if $u.prev = v$, there is an edge $(u, v) \in E$ hence $\{(u, v) | u.prev = v\} \subseteq E$. ◀

The next lemma states that every Node has at most a single *prev* or *tail* pointer pointing at it.

▶ **Lemma 6.** *For every Node u, either tail* $= u$ *and* $|\{v|v.prev = u\}| = 0$ *or* $|\{v|v.prev = u\}| \leq 1$.

Proof. We consider all code lines that assign *prev* pointers. First, notice that the lemma holds in the initial state since only *Nodes*[0] exists and is pointed only by the *tail* pointer.

When a *prev* pointer is assigned at Line 11, it is done using an atomic primitive swap operation, meaning that only a single SWAP operation can read the specific *prev* value that was previously pointed by *tail*.

When a *prev* pointer is assigned during GRECOVER it is done in either Line 48 or 50. In both cases, it is assigned with a Node that starts a path in the Graph(V, E) and by Lemma 5 all Nodes created and *prev* pointers are represented in the graph. It follows that there is no other *prev* pointer pointing to the Node assigned, because it is the start of a maximal path in the graph (V, E). ◀

We now show that the set of paths *Paths* computed in Line 30 is node-disjoint.

▶ **Lemma 7.** *Let P and J be two different paths that exist simultaneously in the global recovery process, and consider a Node* $i \in P$*, then* $i \notin J$.

Proof. From Lemmas 5 and 6, every Node $v \in V$ has at most one incoming edge, meaning there is at most one Node $u \in V$ such that $(u, v) \in E$. In addition, every edge $(u, v) \in E$ signifies that the node representing u has its *prev* pointer pointing to the node representing v. Thus, for every node $u \in V$ there is at most one $v \in V$ such that $(u, v) \in E$. Assume towards a contradiction that there exist two maximal paths P,$J \in Paths$, $P \neq J$ such that $i \in P$ and $i \in J$, then either there are nodes $u \in P$, $j \in J$ s.t $u \neq j$ and $(i, u), (i, j) \in E$, or there are nodes $u \in P$, $j \in J$ s.t $u \neq j$ and $(u, i), (j, i) \in E$. The first case yields a contradiction because node i has two *prev* pointers, and the second case means node i has two prev pointers pointing at it, yielding a contradiction to Lemma 6. ◀

The next lemma shows that after a successful global recovery, the Node list starting at tail is complete and holds all announced Node structures.

▶ **Lemma 8.** *At the end of a crash-free execution of the global GRECOVER procedure, there is a single list induced by prev pointers starting from tail and ending in* $Nodes[0]$ *such that all Node structures announced at Line 9 are in it.*

Proof. First, assume that the condition in Line 38 holds. In this case, there is a maximal *path* $P \in Paths$ that includes both TAILNODE and *Nodes*[0]. Assume towards a contradiction that the lemma does not hold. It follows that there is a Node i that isn't in *path* P. There are two sub-cases to consider. Either there is another *path* J in $Paths$, or there isn't. If the latter sub-case holds, then i must be in $SingleNodes$, hence SWAP will be re-executed starting from Line 10 on its behalf (in Line 39), and because the execution is crash-free, i will be inserted to the list induced by *prev* pointers starting at *tail*, and this list will end in $Nodes[0]$ according to the code. The former sub-case is that $i \in J$ and J is a *middle fragment*, hence it is of length at least two. Let x be the last Node in J and let Op be the operation represented by x. Op must have executed Line 10 before the crash (otherwise no *prev* pointer could point at x) but did not execute Line 11 before the crash (since x is the last node in J). Immediately after Op executed Line 10, *tail* pointed to x. Since Op did not execute Line 11, this contradicts the existence of a path in $Paths$ starting from *tail* and ending in $Nodes[0]$.

Otherwise, the condition in Line 38 does not hold. In this case, it follows from Lines 37–50 that immediately after Line 50 is executed by Op, TailPath is a list that starts from TAILNODE, ends with $Nodes[0]$, and contains all the Nodes in V. Thus by Lemma 5, this list contains all announced Nodes and the lemma holds. ◀

The following lemma ensures that the $\prec$ order can only hold in one direction for any two distinct Paths.

▶ **Lemma 9.** *Let A and B be two paths that exist simultaneously in the global recovery procedure. Then $A \prec B \implies B \not\prec A$.*

Proof. By Definition 4, $A \prec B$ implies that there is a *Node* $x \in A$ and a *Node* $y \in B$ such that $x.endVts < y.startVts$. From Lemma 7, both fragments are disjoint. A new fragment is created only when a process executes Line 10 but does not execute Line 11. This implies that all operations on fragment A that completed Line 10 had done so before any of the operations on fragment B performed Line 10, or vice versa. Since $x.endVts < y.startVts$ and $x.endVts$ is only written after executing Line 10, and since $y.startVts$ is only set before executing Line 10 and after increasing VTS, it follows that all the operations of fragment A executed Line 10 before any the operations of fragment B have executed this line.

If we also have $B \prec A$, then by Definition 4, there is a *Node* $a \in A$, and a *Node* $b \in B$, such that $b.endVts < a.startVts$. It follows that the operation represented by b executed Line 10 before the operation represented by a, which is a contradiction. ◀

The following theorem proves the correctness of our algorithm for the system-wide failures model.

▶ **Theorem 10.** *Algorithm 1 implements a recoverable NRL SWAP in the system-wide failures model using only read, write and primitive swap operations. Its SWAP operations are wait-free and it is strictly recoverable.*

Proof. Clearly from the code, SWAP operations are wait-free since they have no loops, so they terminate in crash-free executions. In addition, SWAP is strictly recoverable as its response is stored in a per-process non-volatile variable Res_i before it returns. We also observe that the algorithm is linearizable in crash-free executions: An operation Op is linearized in Line 10 when it swaps a pointer to its Node to *tail*. The next operation to execute Line 10 after Op (if any) is guaranteed to return Op's input as its response in line 15. If Op is the first SWAP operation to perform Line 10 then, from the initialization of *headNode*, it returns the object's initial value $\perp$.

Next, we consider executions that incur crashes and prove the following property: Upon completion of the GRECOVER procedure, the list $\mathcal{L}$ induced by *prev* pointers, starting from *tail*, contains, exactly once, the Node structure of every SWAP operation announced (in Line 9) in the course of the execution and ends at *headNode*. Moreover, the order induced by $\mathcal{L}$ respects operations' real-time order.

Let $N_1, N_2 = N_1.prev, N_3 = N_2.prev ... N_l = N_{l-1}.prev$ such that $N_1 = Tail$ and $N_l = Nodes[0]$ denote the Node structures in the list induced by *prev* pointers beginning at *tail*. In the initial state, $Tail = N_1 = N_l = Nodes[0]$. For a node x, denote by *proc(x)* the process that created and announced node x and let $H = (SWAP_{proc(N_{l-1})}(N_{l-1}.val), SWAP_{proc(N_{l-2})}(N_{l-2}.val)$ ... $SWAP_{proc(N_1)}(N_1.val))$, where $N_1, \ldots, N_l$ are the nodes in the single fragment that exists immediately after the GRECOVER procedure. Let α denote the execution that ends when GRECOVER completes. From Lemma 8, H is a sequential history that contains all the SWAP operations that were announced in α. Moreover, for any extension β of α in which all these SWAP operations return following the execution of their

individual RECOVER procedures, they return the same values in β and in H. Operations that weren't announced in α but whose individual RECOVER procedures were executed in β, re-execute SWAP(val) (Line 54) and are therefore linearized when they execute Line 10.

It remains to show that for any two announced SWAP operations, SWAP_1 and SWAP_2, if SWAP_1 terminates in α before SWAP_2 starts, then SWAP_1 precedes SWAP_2 in H. Let Node_1, Node_2 be the Nodes created by SWAP_1 and SWAP_2, respectively. Assume towards a contradiction that SWAP_2 precedes SWAP_1 in H. Then there is a path induced by *prev* pointers from Node_1 to Node_2 when GRECOVER terminates. There are two cases to consider. The first case is that all the *prev* pointers of the path between Node_1 and Node_2 were assigned by SWAP operations and not during recovery. This implies that between the time when SWAP_1 executed Line 10 and the time when SWAP_2 executed Line 10, no process performed Line 10 without performing Line 11, otherwise Node_1 and Node_2 would have been on separate fragments just before recovery. Consequently, Node_1 and Node_2 are on the same fragment and Node_1 precedes Node_2 before a crash. It follows that Node_1 precedes Node_2 also in the single fragment that exists when GRECOVER terminates, hence, SWAP_1 precedes SWAP_2 in H, which is a contradiction.

The second case is that the path induced by prev pointers from Node_1 to Node_2 was formed during recovery. There are two sub-cases to consider. The first is that during recovery, one of the Nodes is in *singleNodes* while the other is in a full path from TAILNODE to *Nodes*[0] (thus the condition in Line 38 is true). In this case, since operations that create Nodes in *singleNodes* did not complete, it must be that Node_2 is in *singleNodes* and Node_1 is on the full path. From Line 39, SWAP_2 will be re-executed and so Node_2 will be placed before Node_1 in the list induced by *prev* pointers starting from *tail* at the end of GRECOVER, hence, SWAP_1 precedes SWAP_2 in H, which is a contradiction.

The second sub-case is when the condition of Line 38 is not satisfied. This implies that just before crashing, Node_1 and Node_2 were on different fragments. Let A and B respectively denote the paths representing the fragments on which Node_1 and Node_2 were just before the crash. Since $\text{Node}_1.endVts < \text{Node}_2.startVts$ must hold, from Definition 4, $A \prec B$ holds. Consequently, from Lemma 9 $B \not\prec A$, therefore immediately after GRECOVER terminates there is a path from Node_2 to Node_1, hence, SWAP_1 precedes SWAP_2 in H, which is a contradiction. ◀

4 Detectable Swap Algorithm for the Independent Failures Model

In the independent failures model, each process may crash and recover independently of other processes. A recoverable algorithm for this model must therefore allow one or more processes to execute RECOVER concurrently, while other processes may concurrently execute their SWAP operations. In order to handle this concurrency correctly, we introduce two key changes to Algorithm 2. First, the RECOVER procedure now synchronizes concurrent invocations by using a starvation-free RME lock, implemented from reads and writes only, such as that presented by [20]. This serializes the execution of the recovery code. The goal of the second change is to allow the recovery code to wait for a concurrent SWAP operation *Op* to either complete or crash. Only once this happens, can the recovery code add the Node representing *Op* to graph *G*.

The few additions done in the pseudo-code of SWAP are presented in blue font in Algorithm 1. These consist of adding an *inWork* field (initialized to 0) to the *Node* structure, setting it (in Line 8) just before the SWAP operation is announced in Line 9, and resetting it (in Line 13) immediately after the *endVTS* field is updated in Line 12.

The pseudo-code of the RECOVER procedure for the independent failures model is presented by Algorithm 2. RECOVER first checks whether the *Node* of the crashed operation was announced (Line 3). If it wasn't, it re-executes SWAP(val) (Line 4). Otherwise, it signals that it is performing RECOVER by writing 2 to the *inWork* field of its Node and then attempts to acquire *mutex* (Lines 5–6). Next, it checks if the operation already has a value to return and if so, persists this value and returns it (Lines 7–8, 40–43).

Algorithm 2 Recoverable detectable SWAP, for the independent failures model.

```
1: procedure RECOVER(val)                                   ▷ executed by process i
2:     myNode ← Nodes[i]
3:     if myNode == null or myNode.seq < SEQ_i then
4:         return SWAP(val)
5:     myNode.inWork ← 2
6:     mutex.lock()
7:     if myNode.prev ≠ null then
8:         GoTo Line 40
9:     V_1, E_1 ← gatherGraph()
10:    tailNode ← tail
11:    await(tailNode.inWork ∈ {0, 2})
12:    V_2, E_2 ← gatherGraph()
13:    V_2 ← V_2 ∪ {TAILNODE, tailNode}
14:    E_2 ← E_2 ∪ {(TAILNODE, tailNode)}
15:    V ← V_1 ∪ V_2
16:    E ← E_1 ∪ E_2
17:    Compute set Paths of maximal paths in graph G = (V, E)
18:    for path ∈ Paths do
19:        if myNode ∈ path then
20:            myPath ← path
21:    if len(myPath) == 1 then
22:        re-execute SWAP from Line 10 for myNode
23:        mutex.release()
24:        Res_i ← myNode.prev.val
25:        return Res_i
26:    MiddlePaths ← ∅                                      ▷ May include SingleNodes
27:    for path ∈ Paths do
28:        if TAILNODE ∈ path then
29:            TailPath ← path
30:        else if Nodes[0] ∈ path then
31:            HeadPath ← path
32:        else
33:            MiddlePaths ← MiddlePaths ∪ {path}
34:    ordPaths ← sort(MiddlePaths) in non-increasing ≻ order
35:    candidate ←first path C ∈ ordPaths after myPath s.t. start(C) ∈ V_1 or null if no such C
36:    if candidate ≠ null then
37:        myNode.prev ← start(candidate)
38:    else
39:        myNode.prev ← start(HeadPath)
40:    myNode.endVts ← collect(VTS)
41:    mutex.release()
42:    Res_i ← myNode.prev.val
43:    return Res_i
```

Algorithm 3 The gatherGraph procedure.

```
1: procedure GATHERGRAPH()                                   ▷ Used by Algorithm 2
2:     V ← ∅
3:     E ← ∅
4:     for j from 0 to n do
5:         currNode ← Nodes[j]
6:         while currNode ≠ null do
7:             await(currNode.inWork ∈ {0, 2})
8:             V ← V ∪ {currNode}
9:             if currNode.prev ≠ null then
10:                V ← V ∪ {currNode.prev}
11:                E ← E ∪ {(currNode, currNode.prev)}
12:            currNode ← currNode.prevExecution
```

Lines 9–17 construct the graph $\mathcal{G}$. Unlike the system-wide failure construction, we go over all Nodes *twice*, thus constructing two sets of Nodes, V_1 and V_2. In addition, candidate paths chosen from $MiddlePaths$ are only chosen if their fragment starts from V_1 (Line 35). This is done because, after a single traversal that constructs V_1, there might be a Node in V_1 that is the start of a fragment that may be pointed by some $Node\ x \notin V_1$. As we prove, a second traversal ensures that the problem cannot occur for a graph constructed based on $V = V_1 \cup V_2$. During each traversal of *Nodes*, the algorithm waits for each *Node* v's *inWork* field to be 0 or 2 before adding it to V (Line 7 of *gatherGraph*). This ensures that the operation Op that created v isn't concurrently executing its critical part of SWAP and, therefore, v cannot change after being added to V.

The rest of the procedure is similar to that of GRECOVER in Algorithm 1. All maximal paths in $\mathcal{G}$ are calculated and classified to *TailPath*, *HeadPath* and *MiddlePaths*. In the end a single candidate path either from the sorted *MiddlePaths* or the *HeadPath* is selected to be linked to *myNode* (Lines 34–39). Note that as we ensure only for Nodes in V_1 that any Node pointing at them is in V, only such Nodes are considered as candidates (Line 35). As we prove, this ensures linearizability. The traversals that construct V_1 and V_2 are implemented by the helper function *gatherGraph*, whose pseudo-code is presented by Algorithm 3. The proof of Theorem 11 appears in the full version of this paper.

▶ **Theorem 11.** *Algorithm 2 implements a recoverable NRL SWAP in the independent failures model using only read, write and primitive swap operations and satisfies NRL. Its SWAP operations are wait-free and it is strictly recoverable.*

5 Impossibility of lock-freedom for the independent failures model

In this section, we prove a theorem establishing the impossibility of implementing lock-free algorithms for a wide variety of recoverable objects under the independent failures model. This generalizes previous results [2, 37] to a wider family of operations and implementations. In particular, it applies to any self-implementation of swap under the independent failures model, showing that our usage of a mutual exclusion lock in Algorithm 2 is essential.

We start by defining the notion of a *distinguishable operation*. An operation M is distinguishable, if there exists a history α_{base} in *spec* and two invocations M_p and M_q of M, such that the return values of the invocations allows the system to distinguish which operation is applied right after α_{base}. Formally:

▶ **Definition 12** (Distinguishable operation). *Operation $M : VAL \to RET$ is* distinguishable *if there exists a history α_{base} and values $x, y \in VAL$, $z \in RET$, such that if $M(x)$ and $M(y)$ are applied sequentially right after α_{base}, the first (and only the first) invocation of M to complete returns z.*

Assume a swap object with value 0 and two SWAP operations. If SWAP(1) and SWAP(2) are applied sequentially, only the first operation applied will return 0. This shows that SWAP is a distinguishable operation. Similarly, it's easy to show that pop and deque operations of the stack and queue data structures, as well as *fetch&add* and *test&set*, are also distinguishable operations.

Our impossibility result applies to implementations of distinguishable operations that use only read, write, and a set of *interfering functions*, defined as follows:

▶ **Definition 13** (Interfering functions [22]). *Let F be a set of primitive functions indexed by an arbitrary set K. Define F to be a set of* interfering functions *if for all i and j in K, for any object O that supports f_i and f_j, and for any state S of O, either*

1. *f_i* **and** *f_j* **commute***: The application of f_i to O in state S by process p followed by the application of f_j to O by process q leaves O (but not necessarily the local state of each process) in the same state as the application of f_j to O in state S by process q followed by the application of f_i to O by process p; or*
2. *f_j* **overwrites** *f_i: The application of f_i to O in state S by process p followed by the application of f_j to O by process q leaves O (but not necessarily the local state of each process) in the same state as the application of f_j to O in state S by q alone.*

A *configuration* C consists of the states of all processes and the values of all shared base objects. Sometimes we use the notions of a configuration and a history interchangeably. For example, if a finite history H leads to a configuration C we may use H for representing C when H is clear from the context. Two configurations C_1 and C_2 are *indistinguishable* to a set of processes P, denoted $C_1 \overset{P}{\sim} C_2$, if every process in P has the same state in C_1 and C_2, and every shared object holds the same value in C_1 and C_2.

Given a configuration C reached after a history α_{base}, distinguishable operation M, and a process $r \in \{p, q\}$, we say that C is *r-valent* if there is an execution starting from C in which the return value of M or M.RECOVER by r is z (where α_{base}, M and z are as in Definition 12). C is *bivalent* if it is both p-valent and q-valent, for $p \neq q$. C is *p-univalent* if it is p-valent and not q-valent, and symmetrically for q-univalent. C is *univalent* if it is either p-univalent or q-univalent. Let C be a bivalent configuration and s be a step. If $C \circ s$ is univalent, we say that s is a *critical step*. We generalize the proofs of [2, 37] to prove the following theorem by using valency arguments [15, 18].

▶ **Theorem 14.** *Let M be a distinguishable operation. There is no recoverable implementation I of M from read, write and a set of $K \geq 1$ of interfering primitive operations $f_1 \dots f_K$ in the independent failures model, such that both M and M.RECOVER are lock-free.*

Proof. Assume towards a contradiction that such a lock-free implementation exists. Assume that process p invokes M with value x and process q invokes M with value y, for x, y and z as in Definition 12.

To prove Theorem 14, we construct an execution in which each process performs an infinite number of steps and q neither crashes nor completes its operation.

Configuration C_0, reached after execution α_{base}, is bivalent because a solo execution of either p or q from C_0 returns z. Following a standard valency argument and since we assume that M is lock-free, there is a crash-free execution starting from C_0 that leads to a bivalent

configuration C_1, in which both p and q are about to execute a critical step. It must be that one step leads to a p-univalent configuration while the other leads to a q-univalent configuration. We can prove:

▷ Claim 15. The critical steps of p and q apply (possibly the same) primitives f_i and f_j, respectively, to the same base object.

Proof. Consider all possible steps: read, write, crash and $f_1 \dots f_K$. Assume s_p and s_q are critical steps by process p and q respectively, such that $C_1 \circ s_p$ is p-univalent while $C_1 \circ s_q$ is q-univalent.

- Steps s_p and s_q access distinct registers. In this case, these configurations are indistinguishable to p and q, that is, $C \circ s_p \circ s_q \overset{p,q}{\sim} C \circ s_q \circ s_p$
- Step s_q is a crash step then $C \circ s_p \circ s_q \overset{p}{\sim} C \circ s_q \circ s_p$
- Steps s_p and s_q read the same register. Also in this case $C \circ s_p \circ s_q \overset{p,q}{\sim} C \circ s_q \circ s_p$
- Step s_p writes to some register r step and s_q reads r. In this case, $C \circ s_p \overset{p}{\sim} C \circ s_q \circ s_p$ holds.
- Step s_p applies f_i $1 \leq i \leq K$ and step s_q reads r. In this case, $C \circ s_p \overset{p}{\sim} C \circ s_q \circ s_p$ holds.
- Steps s_p and s_q write to the same register. In this case, $C \circ s_p \overset{p}{\sim} C \circ s_q \circ s_p$ holds.
- Step s_p applies f_i, $1 \leq i \leq K$, step s_q writes to the same register. In this case, $C \circ s_q \overset{q}{\sim} C \circ s_p \circ s_q$ holds.
- Step s_p applies f_i, $1 \leq i \leq K$, step s_q applies f_j, $1 \leq j \leq K$ each to a different base object O, In this case $C \circ s_p \circ s_q \overset{p,q}{\sim} C \circ s_q \circ s_p$ holds.

In each of the above cases, the configurations are indistinguishable to at least one process, and therefore, must have the same valencies. Therefore, it must be that p and q apply f_i and f_j respectively to the same base object. ◁

Assume, without loss of generality, that $C_1 \circ p$ is p-univalent while $C_1 \circ q$ is q-univalent. We consider two cases:

Case 1: f_i and f_j commute. Consider executions C_2, C_3 where $C_2 = C_1 \circ p \circ q \circ CRASH_p$ and $C_3 = C_1 \circ q \circ p \circ CRASH_p$. Configurations C_2 and C_3 are reached after p and q each take a step (in different orders) in which they apply their operations to the same base object O and then p crashes.

A solo execution of M.RECOVER by p from both C_2 and C_3 must complete since I is lock-free. Furthermore, $C_2 \overset{p}{\sim} C_3$ holds, because p's response from the primitive f_i is lost, while the value of O is the same in both configurations since f_i and f_j commute. Consequently, an execution of M.RECOVER by p from both C_2 and C_3 must return the same value. Let v denote this value.

Assume first that $v = z$ and thus C_3 is p-valent. Configuration $C_1 \circ q \circ p$ is q-univalent, while $C_3 = C_1 \circ q \circ p \circ CRASH_p$ is p-valent. However, $C_1 \circ q \circ p \overset{q}{\sim} C_3$ holds because q is unaware of p's crash. Consequently, a solo execution of q from C_3 must return z, that is, C_3 is also q-valent. This proves that C_3 is bivalent.

Assume then that $v \neq z$. We now show that, in this case, C_2 is bivalent. Indeed, from this assumption, C_2 is q-valent, because a solo execution of q after p completes (and returns $v \neq z$) must return z since, from Definition 12, exactly one of these two operations must return z. However, configuration $C_1 \circ p \circ q$ is p-univalent, while $C_2 = C_1 \circ p \circ q \circ CRASH_p \overset{q}{\sim} C_1 \circ p \circ q$, therefore a solo execution of q from C_2 must return x s.t. $x \neq z$. Thus, C_2 is bivalent.

Case 2: f_j "overwrites" f_i. Consider executions C_2, C_3 where $C_2 = C_1 \circ p \circ q \circ CRASH_p$ and $C_3 = C_1 \circ q \circ CRASH_p$. A solo execution of M.RECOVER by p from both C_2 and C_3 must complete since I is lock-free. Furthermore, $C_2 \overset{p}{\sim} C_3$ because p's response from the primitive f_i is lost, while the value of the base object f_i and f_j are applied to is the same in both configurations since f_j "overwrites" f_i. Therefore, an execution of M.RECOVER by p from both C_2 and C_3 returns the same value. Let v denote this value.

Assume $v = z$ and thus C_3 is p-valent. $C_1 \circ q$ is q-univalent, while $C_3 = C_1 \circ q \circ CRASH_p$ is p-valent. $C_1 \circ q \overset{q}{\sim} C_3$ holds because q is unaware of p's crash. Therefore, a solo execution of q from C_3 returns z, that is, C_3 is also q-valent. This proves that C_3 is bivalent.

Assume then that $v \neq z$. We show that in this case C_2 is bivalent. Indeed, from our assumption, C_2 is q-valent, as a solo execution of q after p completes must return z since, from Definition 12, exactly one of these two operations must return z. However configuration $C_1 \circ p \circ q$ is p-univalent and $C_2 = C_1 \circ p \circ q \circ CRASH_p \overset{q}{\sim} C_1 \circ p \circ q$, therefore a solo execution of q from C_2 must return $x \neq z$. This establishes that C_2 is bivalent.

In both cases, this shows that we can keep extending the execution obtaining an infinite execution in which neither p nor q complete their operations and q performs an infinite number of steps without crashing, contradicting the lock-freedom assumption. ◀

Golab proves for the independent failures model that any commutative/overwriting primitive with consensus number 2 drops to level 1 in the recoverable consensus hierarchy [18, Theorem 4.9]. Although the model he assumes differs from ours (e.g., there is no separation between an operation and its recovery code) and the problems for which the impossibility results hold are different as well, our proof is similar to his. It might be possible to prove Theorem 14 by using a reduction from [18, Theorem 4.9]. However, we think that our simple direct proof makes the presentation clearer and more self-contained.

6 Discussion

We present two NRL self-implementations of the swap object, one for the system-wide failures model and the other for the independent failures model. In both, SWAP operations are wait-free and the recovery code is blocking. In the system-wide failures model, this is a result of delegating the recovery to a single process, while in the independent failures model, it is due to coordination between the recovering process and the other processes. Both our algorithms support a strictly recoverable SWAP operation. We also prove the impossibility of a lock-free implementation of distinguishable operations using read-write and a set of interfering functions, in the independent failures model. In particular, this shows that with independent failures, a self-implementation of swap cannot be lock-free.

Our algorithms use $O(m * n)$ space, where m is the number of SWAP invocations in the execution. Bounding memory consumption to $O(n)$ is relatively easy if a recoverable SWAP operation by one process can wait for operations by other processes to either make progress or fail. An interesting open question is to figure out whether the space complexity of detectable swap self-implementations with wait-free operations can be reduced to $o(m)$ or if $\Omega(m)$ is inherently required. We leave this question for future work.

Finally, it is also important to explore how self-implementations, in particular of SWAP, can be used to turn non-recoverable higher-level objects into NRL implementations of the same objects.

References

1. Marcos K Aguilera and Svend Frølund. Strict linearizability and the power of aborting. *Technical Report HPL-2003-241*, 2003.
2. Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Nesting-safe recoverable linearizability: Modular constructions for non-volatile memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 7–16, 2018. doi:10.1145/3212734.3212753.
3. Ohad Ben-Baruch, Danny Hendler, and Matan Rusanovsky. Upper and lower bounds on the space complexity of detectable objects. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, pages 11–20, 2020. doi:10.1145/3382734.3405725.
4. Naama Ben-David, Guy E Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 253–264, 2019. doi:10.1145/3323165.3323187.
5. Naama Ben-David, Michal Friedman, and Yuanhao Wei. Survey of persistent memory correctness conditions. *arXiv preprint*, 2022. arXiv:2208.11114.
6. Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In *19th International Conference on Principles of Distributed Systems (OPODIS 2015)*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPICS.OPODIS.2015.20.
7. Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. *ACM SIGPLAN Notices*, 49(10):433–452, 2014. doi:10.1145/2660193.2660224.
8. David Yu Cheng Chan and Philipp Woelfel. Recoverable mutual exclusion with constant amortized RMR complexity from standard primitives. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, pages 181–190, 2020. doi:10.1145/3382734.3405736.
9. Kyeongmin Cho, Seungmin Jeon, and Jeehoon Kang. Practical detectability for persistent lock-free data structures. *arXiv preprint*, 2022. arXiv:2203.07621.
10. Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. The inherent cost of remembering consistently. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 259–269, 2018. doi:10.1145/3210377.3210400.
11. Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 271–282, 2018. doi:10.1145/3210377.3210392.
12. Andreia Correia, Pascal Felber, and Pedro Ramalhete. Persistent memory and the rise of universal constructions. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020. doi:10.1145/3342195.3387515.
13. Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. In *2018 USENIX Annual Technical Conference*, pages 373–386, 2018. URL: https://www.usenix.org/conference/atc18/presentation/david.
14. Panagiota Fatourou, Nikolaos D Kallimanis, and Eleftherios Kosmas. The performance power of software combining in persistence. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 337–352, 2022. doi:10.1145/3503221.3508426.
15. Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985. doi:10.1145/3149.214121.
16. Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E Blelloch, and Erez Petrank. Nvtraverse: In nvram data structures, the destination is more important than the journey. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 377–392, 2020. doi:10.1145/3385412.3386031.
17. Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. *ACM SIGPLAN Notices*, 53(1):28–40, 2018. doi:10.1145/3178487.3178490.

18 Wojciech Golab. The recoverable consensus hierarchy. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 281–291, 2020. doi:10.1145/3350755.3400212.

19 Wojciech Golab and Danny Hendler. Recoverable mutual exclusion in sub-logarithmic time. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 211–220, 2017. doi:10.1145/3087801.3087819.

20 Wojciech Golab and Aditya Ramaraju. Recoverable mutual exclusion. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 65–74, 2016. doi:10.1145/2933057.2933087.

21 Rachid Guerraoui and Ron R Levy. Robust emulations of shared memory in a crash-recovery model. In *24th International Conference on Distributed Computing Systems, 2004. Proceedings.*, pages 400–407. IEEE, 2004. doi:10.1109/ICDCS.2004.1281605.

22 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991. doi:10.1145/114005.102808.

23 Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990. doi:10.1145/78969.78972.

24 Morteza Hoseinzadeh and Steven Swanson. Corundum: Statically-enforced persistent memory safety. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 429–442, 2021. doi:10.1145/3445814.3446710.

25 Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via justdo logging. *ACM SIGARCH Computer Architecture News*, 44(2):427–442, 2016. doi:10.1145/2872362.2872410.

26 Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Distributed Computing: 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings 30*, pages 313–327. Springer, 2016. doi:10.1007/978-3-662-53426-7_23.

27 Prasad Jayanti, Siddhartha Jayanti, and Sucharita Jayanti. Durable algorithms for writable ll/sc and cas with dynamic joining. *arXiv preprint*, 2023. arXiv:2302.00135.

28 Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. Optimal recoverable mutual exclusion using only fasas. In *Networked Systems: 6th International Conference, NETYS 2018, Essaouira, Morocco, May 9–11, 2018, Revised Selected Papers*, pages 191–206. Springer, 2019. doi:10.1007/978-3-030-05529-5_13.

29 Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. A recoverable mutex algorithm with sub-logarithmic RMR on both CC and DSM. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 177–186, 2019. doi:10.1145/3293611.3331634.

30 Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. Constant RMR recoverable mutex under system-wide crashes. *arXiv preprint*, 2023. arXiv:2302.00748.

31 Prasad Jayanti and Anup Joshi. Recoverable FCFS mutual exclusion with wait-free recovery. In *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPICS.DISC.2017.30.

32 Prasad Jayanti and Anup Joshi. Recoverable mutual exclusion with abortability. *Computing*, 104(10):2225–2252, 2022. doi:10.1007/S00607-022-01105-1.

33 Daniel Katzan and Adam Morrison. Recoverable, abortable, and adaptive mutual exclusion with sublogarithmic RMR complexity. In *24th International Conference on Principles of Distributed Systems*, 2021. doi:10.4230/LIPICS.OPODIS.2020.15.

34 Nan Li and Wojciech Golab. Detectable sequential specifications for recoverable shared objects. In *35th International Symposium on Distributed Computing (DISC 2021)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.DISC.2021.29.

35 Virendra Marathe, Achin Mishra, Amee Trivedi, Yihe Huang, Faisal Zaghloul, Sanidhya Kashyap, Margo Seltzer, Tim Harris, Steve Byan, Bill Bridge, et al. Persistent memory transactions. *arXiv preprint*, 2018. arXiv:1804.00701.

36 John M Mellor-Crummey and Michael L Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991. doi:10.1145/103727.103729.

37 Liad Nahum, Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Recoverable and detectable fetch&add. In *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.OPODIS.2021.29.

38 Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B Morrey III, Dhruva R Chakrabarti, and Michael L Scott. Dalí: A periodically persistent hash map. In *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPICS.DISC.2017.37.

39 Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. Onefile: A wait-free persistent transactional memory. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 151–163. IEEE, 2019. doi:10.1109/DSN.2019.00028.

40 Matan Rusanovsky, Hagit Attiya, Ohad Ben-Baruch, Tom Gerby, Danny Hendler, and Pedro Ramalhete. Flat-combining-based persistent data structures for non-volatile memory. In *Stabilization, Safety, and Security of Distributed Systems: 23rd International Symposium, SSS 2021, Virtual Event, November 17–20, 2021, Proceedings 23*, pages 505–509. Springer, 2021. doi:10.1007/978-3-030-91081-5_38.

41 David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. NVC-hashmap: A persistent and concurrent hashmap for non-volatile memories. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics*, pages 1–8, 2015. doi:10.1145/2803140.2803144.

42 Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. Efficient lock-free durable sets. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–26, 2019. doi:10.1145/3360554.