Theoretical Computer Science 503 (2013) 89–108

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

On the power of breakable objects ☆

Wei Chen [a], Guangda Hu [b], Jialin Zhang [c,*]

[a] Microsoft Research Asia, Beijing, China
[b] Princeton University, New Jersey, USA
[c] Institute of Computing Technology, Chinese Academy of Sciences, Kexueyuan South Road, Haidian, Beijing, China

ARTICLE INFO

Article history:
Received 24 November 2011
Received in revised form 10 January 2013
Accepted 31 May 2013
Communicated by M. Mavronicolas

Keywords:
Distributed computing
Shared memory
Wait-free hierarchy
Consensus number
Breakable objects

ABSTRACT

In distributed shared-memory systems, a *breakable object* is one that may enter a special *broken* state, after which all operations will fail and return a special broken symbol $\perp_B$. Most breakable objects appeared in literature belong to a class we call *acyclic* breakable objects, in which every operation moves the object closer to the broken state. We show in general that acyclic breakable objects have limited consensus power. Our main results focus on the richer but less covered *cyclic* breakable objects. We study the consensus numbers of cyclic breakable objects by using a general algorithmic framework, and obtain several interesting results. In particular one result on a breakable queue points out a mistake that appeared both in a previous paper and a textbook. We further study *operation-wise* breakable objects in which some operations on the object may break but not other operations. We use various breakable queues as running examples, and demonstrate that the generic framework can be applied in this context as well, and in some cases with nontrivial implementations of the framework. We also provide matching impossibility results to obtain exact consensus numbers for all the cases we consider. Finally, since all example objects we found in the literature with a generic consensus number n are breakable objects, we provide a simple and natural non-breakable object class with generic consensus number n.

1. Introduction

In shared-memory distributed systems, individual processes communicate with one another through various shared objects, such as read-write registers, queues, stacks, and testAndSet. In [4], Herlihy introduces consensus numbers to compare the power of various shared objects in standard asynchronous and wait-free shared-memory distributed computing model. Consensus is a distributed task in which processes propose their values and they need to agree on one irrevocable value from one of the values proposed, and the decision should be made no matter how many processes participate in consensus. A shared object type has consensus number n if using any number of objects of this type plus any number of read-write registers one can implement n-process consensus, but not $(n+1)$-process consensus. Herlihy uses consensus numbers to build a wait-free hierarchy, in which different objects are placed at different levels. For example, read-write registers are the weakest, with consensus number 1; normal queues, stacks, and testAndSet objects have consensus number 2; while queues augmented with a peek operation have consensus number infinity, meaning that they can be used to solve consensus with any number of processes.

☆ This work was supported in part by the National Natural Science Foundation of China Grants 61170062, 61222202.
* Corresponding author.
E-mail addresses: weic@microsoft.com (W. Chen), guangdah@cs.princeton.edu (G. Hu), zhangjl2002@gmail.com (J. Zhang).

0304-3975/$ – see front matter
http://dx.doi.org/10.1016/j.tcs.2013.05.036

Table 1
Consensus numbers of various breakable queues.

Enqueue operation	Dequeue operation		
	Normal	Break-all	Break-op
Normal	2 (normal queue, [4])	∞ (Theorem 4)	∞ (Theorem 7)
n-cell normal	2 (similar to the above)	∞ (Theorem 4)	n (Theorems 8, 9)***
n-cell break-all	∞ (n-BQ, Theorem 3)†	$n+1$ (Theorems 5, 6)	$2n$ (Theorems 13, 14)
n-cell break-op	$2n$ (Theorems 11, 12)*	$2n$ (Theorems 15, 16)*	n (Theorems 8, 10)**

Roughly, "normal" means the operation will never fail. "break-all" means after the operation fails, all subsequent operations on the object will fail, while "break-op" means only subsequent operations of the same type will fail. See Section 4 for details.

* Correct for $n \geqslant 2$; when $n = 1$, consensus number is 3 (Theorem 18).

** Correct for $n \geqslant 3$; when $n = 1$ or 2, consensus number is 3 (Theorems 18, 19).

*** Correct for $n \geqslant 2$; when $n = 1$, consensus number is 2 (by an argument similar to the normal queue).

Others are correct for $n \geqslant 1$.

† This result fixes a mistake appearing in [6] (a claim after its Claim 5), and in [2], hint of Exercise 15.8.

Objects studied in [4] are all objects that will not be broken under any circumstances. Later, in [6] Jayanti and Toueg introduce an object that may break under certain conditions. Their purpose is to provide a class of objects with consensus number n for any positive integer n. The object class they introduced is called n-bounded peek queue, which is based on a queue with enqueue and peek operations (but no dequeue operations) and has at most n cells. When the queue is full, the next enqueue operation will break the queue and all subsequent operations including this enqueue will return a broken symbol $\perp_B$. Breakable objects are also used in textbooks [2,5] as examples of shared objects with consensus number n. In [1], Afek and Shalom study bounded-use objects, in which certain operations can only be invoked for a bounded number of times and then fail. This is another kind of objects that can break certain operations but not others.

In this paper, we provide a systematic study on the consensus power of objects that may break or objects having some operations that may break, which we refer to as *breakable* objects. We found that almost all breakable objects studied before [1,5,6] are restricted to one class of objects — objects which are unidirectionally moving towards their broken states. For example, n-bounded peek queue of [6] always moves towards filling the queue and then breaks (since there is no dequeue operation). We call these objects *acyclic* and show that their consensus power are limited by the distance between their normal states and the broken state.

Our main focus of the paper is on the richer class of *cyclic* objects whose operations may move their states away from the broken states, e.g., adding the dequeue operation to the n-bounded peek queue. We provide a general algorithmic framework and show how to implement n-process consensus in this framework. All actual consensus algorithms are various implementations of this framework using different breakable objects.

We use various breakable queues as running examples to show how to apply the generic framework. Our results are summarized in Table 1. First, we study objects that have an object-wide broken state such that all operations are broken once in this state. One important result in this category is that, if the queue has n cells, a normal dequeue operation and an enqueue operation that will break the queue when the queue is already full (*n-cell break-all enqueue*), its consensus number is infinity. This result corrects a mistake appearing both in [6], which claimed that even adding the dequeue operation to n-bounded peek queue the consensus number is still n, and in [2], Exercise 15.8, which hinted that n-bounded queue augmented with the peek operation has consensus number n. More importantly, this result illustrates the drastic difference between acyclic breakable objects and cyclic ones: n-bounded peek queue is acyclic and has consensus number n, but once adding the dequeue operation, it is cyclic and its consensus power increases to infinity. We also study queues that may break when dequeuing an empty queue (*break-all* dequeue operation), and the combination of break-all dequeue and n-cell break-all enqueue operations. The former has consensus number infinity, while the latter has consensus number $n+1$.

Next, we study *operation-wise breakable objects* with *break-op* operations, which are operations that may break under certain state but not breaking the object itself or other operations. In the case of queues, a break-op dequeue operation will break all subsequent dequeues when the queue is empty (but not breaking the enqueues), while an n-cell break-op enqueue operation will break all subsequent enqueues after doing enqueue operation in a full queue (but not breaking the dequeue operations). We study combinations of break-op operations with normal operations or break-all operations, and fully characterize their consensus numbers for all variants of queues (Table 1). Our algorithmic results are still based on the generic framework, but we need more involved implementations of the framework using objects with break-op operations. All our impossibility results apply the bivalency argument [4], but some need involved case studies to deal with certain tricky cases. The results themselves are also interesting. For example, an n-cell queue with a break-op dequeue operation and a normal enqueue operation has consensus number n, but the seemingly symmetric object with a break-op enqueue and a normal dequeue has consensus number $2n$. This illustrates subtle differences between dequeue and enqueue operations. From the results and their analyses, we see that intuitively the difference in the consensus power of these objects is due to the ability of communicating about object break status among processes.

The above results are not specific to the breakable queue objects. We point out that most of our algorithmic results rely on the state transition patterns of the breakable object, not on specific value domain or the semantics of the operations. For example, an n-bounded counter has the same state transition pattern as an n-bounded queue, and thus the two have the

same consensus numbers. Moreover, all our algorithmic results on breakable queues are easily carried over to corresponding breakable stacks.

Finally, we propose a new non-breakable object n-bounded shift augment queue, which has consensus number n. We notice that all objects having a generic consensus number n in the literature are breakable objects. Thus this new object could be a nice addition to the literature as the first natural non-breakable object with a generic consensus number n.

As a summary, we conduct a systematic study of the consensus power of breakable objects, especially of cyclic breakable objects that were not covered well before. We correct a mistake on breakable queues in the literature, and provide a comprehensive characterization of various breakable queues. Our study enriches the theory of wait-free consensus hierarchy by introducing a large class of breakable objects and a generic framework applicable to these objects. It also provides rich pedagogical materials for teaching the consensus power of shared objects. It may also has practical implementations, since it provides a new class of simple and universal shared objects that could be used to implement any other objects in wait-free systems.

1.1. Related work

The first breakable object seems to be the n-bounded peek queue in [6]. Breakable objects also appear in textbooks [2,5], and in the context of studying bounded-use and bounded-size objects [1]. Although the study in [1] bears some resemblance with our work, introducing broken states is different from bounded-use and bounded-size restrictions, and it leads to several important differences between the two works. First, their bounded-use objects are limited cases of our operation-wise breakable objects. In our case, the number of operation invocations are not bounded, and no operation will break as long as the object is in safe states. We believe this makes significant difference in the consensus power of objects. Second, their bounded-size objects do not have broken states, so they are different from our bounded-size breakable objects. Third, in our work we are able to provide a general algorithmic framework, and we also point out that the consensus power of breakable objects are mainly due to their general state transition patterns. This seems not the case for the bounded-use objects — their algorithms are individually designed for each object and rely on detailed semantics of the objects.

Some other objects in the literature may bear resemblance with breakable objects. For example, sticky bit [7] is an object such that the first value written to the object sticks. It can be viewed as later writes are broken. This is a special case of bounded-use objects, and thus has the same limitation as discussed above.

Paper organization. In Section 2, we provide basic definitions and terminologies of the paper. In Section 3, we show the general results on breakable objects, including the upper bound on the consensus number of acyclic breakable objects, and the generic algorithmic framework implementing consensus using cyclic breakable objects. Sections 4 and 5 provide results on object-wide breakable queues and operation-wise breakable queues, respectively. In particular, all lower bound results on consensus numbers are applications of the generic algorithmic framework, with matching upper bounds proven by the bivalency arguments. In Section 6, we provide the first object type with no broken states and a generic consensus number n. We conclude the paper in Section 7.

2. Preliminaries

We consider a standard asynchronous shared-memory distributed computing model as given in [4]. The system is equipped with shared-memory objects, each of which has a certain *type*. An object type $T = (\Sigma_T, \Omega_T)$ specifies (a) Σ_T, the set of possible states of the object, and (b) Ω_T, the set of possible primitive operations on the object, for example, *deq* and $enq(v)$ operations of queue object. The argument (if any) is part of the operation, for example, two *enq* operations with different arguments are two different operations. Each operation $\omega \in \Omega_T$ of the object is deterministic, and thus is modeled as a function mapping the current state of the object to a new state of the object and the return value of the operation (could be null). For example, *register* is a basic type with its state coming from a value domain V plus a special empty symbol $\perp$, and operations *read* and $write(v)$'s with standard semantics. The main class of objects we study in this paper are queues. A normal *queue* type has a sequence of values (of certain value domain V) as its states, and has a dequeue operation *deq* and enqueue operations $enq(v)$ for all values $v \in V$. Operations *deq* and $enq(v)$ have standard semantics: *deq* removes the head of queue and returns it, and returns $\perp_E$ if the queue is empty, and $enq(v)$ appends v to the end of queue with default return value.

A *concurrent system* consists of a finite number of processes and a number of base shared-memory objects, such that processes communicate with one another solely via accessing these base objects. Every operation by a process on a base object is atomic, that is, in one atomic step, one process can invoke one operation on one object and obtain its return value, and based on the return value and the local state of the process conduct a state transition to reach a new process state. We consider deterministic algorithms in the concurrent system, which on each process p at every state s, specify the object o and the operation ω on o to be applied, and also specify the new state s' based on the current state s and the return value v of the operation ω on o. A *configuration* C is a collection of process states and object states, reflecting the state of the system at a point in time. A run of an algorithm is an infinite sequence $C_0, p_1, C_1, p_2, \ldots$, starting with the initial configuration C_0 and alternating between configurations and process identifiers. In the sequence, configuration C_i is the

result of process p_i taking a step at configuration C_{i-1}, for all $i \geqslant 1$. Process run is asynchronous, meaning that between any two steps of a process there could be an arbitrary number of steps of other processes. A *correct* process in a run is one that executes an infinite number of steps in the run, and a *faulty* process is one that executes only a finite number of steps.

In this paper, we study the power of shared objects by their ability of implementing consensus tasks [4], as defined below. In consensus, each process proposes a value $v \in V$ from some value domain V by invoking *consensus*(v), and the return value is called its consensus decision. It needs to satisfy the following properties:

- *Agreement*: no two processes decide differently.
- *Termination*: if a correct process proposes, eventually it decides.
- *Validity*: if a process decides v, then v has been proposed by some process.

Note that the Termination property implies that it is *wait-free*, that is, the process will terminate its consensus no matter if other processes participate or stop participating on the consensus task. We say that an object type T solves n-process consensus if there is an algorithm implementing consensus among n processes using any number of registers and objects of type T. We say that object type T has *consensus number* n if it could solve n-process consensus but not $(n+1)$-process consensus, and it has consensus number infinity if it could solve n-process consensus for all $n \in \mathbb{N}$.

In this paper, we focus on objects that may break and end up in a broken state. Technically, for an object type $T = (\Sigma_T, \Omega_T)$, we say that a set of states $B \subseteq \Sigma_T$ are *broken* states of the object if (a) any operation $w \in \Omega_T$ applied at state $b \in B$ keeps the object state in B and returns a special predefined symbol $\perp_B$; and (b) for any operation $w \in \Omega_T$ on a state $s \in \Sigma_T$ that does not result in a state $b \in B$, its return value is not $\perp_B$. Since there is no need to distinguish states in B, we collapse all states in B into one special state s_B, which is called the *broken* state of the object. An object with a broken state is called an *object-wide breakable* object, and after the object enters its broken state, we say that it is *broken*. For example, an *n-bounded queue* type [6,2], denoted as n-BQ, is a queue with *deq* and *enq*(v) operations and only n cells, such that when the queue is full (all n cells contain previously enqueued values), the next *enq*$(\cdot)$ operation will break the queue and return $\perp_B$, and all subsequent operations will keep the queue in the broken state and return $\perp_B$. A related object is *n-bounded peek queue* [6], denoted as n-BPQ, which differs from n-BQ by removing the *deq* operation but adding the *peek* operation, which returns the head of queue without changing the queue state.

For a breakable object type $T = (\Sigma_T, \Omega_T)$, we say that a state $c \in \Sigma_T$ is a *cliff* state of T if there exists an operation $\omega \in \Omega_T$ such that ω applied to c results in the broken state s_B. We refer to operation ω at c as a *jump* operation. We exclude trivial breakable objects that either contain only the broken state or contain no transition from non-broken states to the broken state, and thus all breakable objects contain cliff states. If a breakable object has a state s that is neither a broken state or a cliff state, we call it a *safe* state. An operation that transfers a cliff state to a safe state is called a *save* operation. For example, for the n-bounded queue type, any full queue state is a cliff state, any *enq*(v) operation on the full queue is a jump operation, any non-full queue state is a safe state, and the *deq* operation at any full queue state is a save operation.

We say that an operation is *trivial* if it never changes the state of the object. Given an object type $T = (\Sigma_T, \Omega_T)$, its state transition graph G_T is a labeled graph such that the vertex set is Σ_T, and for every state $s \in \Sigma_T$ and every operation $\omega \in \Omega_T$, there is an edge from s to s' labeled with ω, where s' is the resulting state when applying ω to s. The transition graph allows self-loops and multi-edges. A breakable object type T is *acyclic* if the subgraph of G_T after removing the broken state and all self-loops labeled by trivial operations is a directed acyclic graph (DAG). A breakable object type T is *cyclic* if it is not acyclic. For example, n-bounded queue is cyclic while n-bounded peek queue is acyclic. In particular, the latter has no save operations. We will see that whether the breakable object is cyclic or not could make a big difference in its consensus power.

We also study partially breakable objects. In particular, a set of operations $A \in \Omega_T$ is breakable if for any operation $\omega \in A$, there exists some state $s \in \Sigma_T$ such that ω applied to s leaves the state unchanged and returns the special symbol $\perp_B$; and (b) once $\perp_B$ is returned by some $\omega \in A$, any subsequent invocation of any operation in A in the run will not change the object state and will always return $\perp_B$. We call objects with breakable operations *operation-wise* breakable objects. Note that when some set of operations break, the object itself is not broken, and other operations not in A may continue to behave normally. For example, an *n-bounded enq-breakable queue* is a queue with at most n cells. When the queue is full, the next *enq*(v) operation for any $v \in V$ will break and all subsequent *enq*(v) operations for any $v \in V$ will only return $\perp_B$, but the *deq* operation can continue behaving normally. Bounded-use objects studied in [1] can be viewed as a special type of operation-wise breakable objects, which break an operation after a certain number of invocations of the operation.

3. General results on breakable objects

In this section, we first show that acyclic breakable objects have limited consensus power, and then provide an algorithmic framework to solve consensus based on cyclic breakable objects. As we shall see, our results are based on the state transition pattern among cliff, safe, and broken states of a breakable object, not on the actual values stored in the objects or semantics of their operations. Thus, they are generic results illustrating the power of breakable objects.

3.1. Limited consensus power of acyclic breakable objects

As pointed out in the introduction, most of the breakable objects encountered in the literature are acyclic breakable objects. In this section, we show that their consensus power is limited by the diameter of their state transition graphs. This serves as a motivation for us to turn our attention to the less studied cyclic breakable objects.

For impossibility results, it is enough to show that binary consensus is not solvable. For binary consensus, we define configuration valency as the standard one [3]: a configuration is *v-valent* (v is 0 or 1) if all possible runs from it reach decision v; a configuration is *univalent* if it is 0-valent or 1-valent; and a configuration is *bivalent* if it is not univalent. All impossibility results of this paper will use the following lemma, which is essentially proven in [4] but not explicitly summarized as a lemma there.

Lemma 1. *(See [4].) If an object type T solves n-process consensus for some $n > 1$, then there is a bivalent critical configuration C reachable from some initial configuration satisfying the following properties*: (a) *every operation of every process on C results in a univalent configuration*; (b) *all operations on C must be on the same object of type T (not a register)*; *and* (c) *none of the operations on C is trivial.*

The following theorem provides an upper bound on the consensus number of acyclic breakable objects.

Theorem 1. *For an acyclic breakable object type T with state transition graph G_T, let G'_T be the acyclic subgraph of G_T after removing the broken state and the self-loops labeled by trivial operations. If the length of the longest path in G'_T is n, then the consensus number of T is at most $n+1$. Moreover, if all of its jump operations on cliff states return $\perp_B$ and $n \geqslant 2$, its consensus number is at most n.*

Proof. Assume the object type T can solve $(n+2)$-process consensus ($n \geqslant 1$). By Lemma 1, there is a bivalent critical configuration C satisfying the three properties stated in Lemma 1. Define $\mathcal{S}_1$ to be the set of processes whose next operation leads to a 1-valent configuration, and $\mathcal{S}_0$ to be the set of processes whose next operation leads to a 0-valent configuration. By property (a), the union of $\mathcal{S}_1$ and $\mathcal{S}_0$ is the set of all processes, hence $|\mathcal{S}_1| + |\mathcal{S}_0| = n+2 \geqslant 3$. Without loss of generality we assume $|\mathcal{S}_1| \geqslant 2$. Pick a process P_1 in $\mathcal{S}_1$, consider the following two sequences of steps from C ($\mathcal{S}_1 \setminus \{P_1\} \neq \emptyset$):

1. Every process in $\mathcal{S}_1 \setminus \{P_1\}$ takes an operation, then every process in $\mathcal{S}_0$ takes an operation;
2. Every process in $\mathcal{S}_0$ takes an operation, then every process in $\mathcal{S}_1 \setminus \{P_1\}$ takes an operation.

The first sequence leads to a 1-valent configuration C_1 but the second leads to a 0-valent configuration C_0. By properties (b) and (c), both sequences take $n+1$ nontrivial operations on a single object. As the longest path in the DAG G'_T has length n, the object must be in the broken state after either step sequence. Now if P_1 runs solo after C_0 or C_1, it cannot distinguish these two configurations and thus must decide on the same value. This contradicts the fact that C_0 is 0-valent and C_1 is 1-valent. Therefore the consensus number of T is at most $n+1$.

If all of T's jump operations on cliff states return $\perp_B$ and $n \geqslant 2$, we show that the consensus number is at most n. Similarly we can define the two sets $\mathcal{S}_1$ and $\mathcal{S}_0$, and we have $|\mathcal{S}_1| + |\mathcal{S}_0| = n+1 \geqslant 3$. Assume $|\mathcal{S}_1| \geqslant 2$, pick a process $P_1 \in \mathcal{S}_1$ and consider the two step sequences above. Both sequences take n nontrivial operations on a single object. This object may be in a cliff state or a broken state. Now if P_1 runs solo, its first operation returns $\perp_B$ and then the object must be in a broken state. P_1 cannot distinguish these two runs from now on and has to decide on the same value — a contradiction. Therefore the consensus number of T is at most n. □

The above theorem immediately implies that n-bounded peek queue [6] and n-bounded compareAndSet (Exercise 5.70 of [5]) with $n \geqslant 2$ have consensus number at most n, because for both objects, it only takes at most n nontrivial operations to transfer the object to the cliff state, and the next nontrivial operation is a jump operation with return value $\perp_B$.[1] The theorem indicates that purely by the state transition pattern we can upper bound the consensus power of acyclic breakable objects. This leads us to look into cyclic breakable objects and see if they can provide higher consensus power. The rest of the paper thus focuses on cyclic breakable objects.

3.2. Algorithmic framework for consensus implementation using cyclic breakable objects

We develop a general algorithmic framework to solve consensus based on state transition patterns of cyclic breakable objects with save operations. The idea is to use the differentiation power of the cliff states — if applying a jump operation at a cliff state, it will go to the broken state, but if applying a save operation, it will go to a safe state. All implementations in this paper on actual breakable objects are different realizations of the generic framework.

[1] Note that when $n = 1$, both n-bounded peek queue and n-bounded compareAndSet have consensus number 2 instead of 1, since both can easily implement testAndSet objects. This subtle difference for the case of $n = 1$ was not pointed out in [6] for n-bounded peek queue, or in [5], Exercise 5.70 for n-bounded compareAndSet.

In the general framework, we have two (possibly non-atomic) methods, *jumpAndCheck* and *saveAndCheck*, each of which is implemented by atomic operations on some base breakable object of type T.[2] Both methods have two possible return values — BROKEN or SAFE. We say that a method invocation I_1 is before a method invocation I_2 if the first atomic operation in I_1 is executed before any atomic operation in invocation I_2. We also have an initial state of the object s_0. We require that methods *jumpAndCheck* and *saveAndCheck* together with the initial state s_0 satisfy the following condition:

- *n-Break Condition*: If there is at most one invocation of method *jumpAndCheck* and at most $n-1$ invocations of method *saveAndCheck* on an object with initial state s_0, then all method invocations return the same symbol, and they return the BROKEN symbol if and only if *jumpAndCheck* is invoked before any other method invocations.

If methods *jumpAndCheck* and *saveAndCheck* together with an initial state s_0 satisfy the n-Break Condition, then we could use them to implement n-process consensus. Algorithm 1 provides the pseudocode for it among processes $\{P_1, P_2, \ldots, P_n\}$. It uses *jumpAndCheck* and *saveAndCheck* on n breakable objects $\{O_1, O_2, \ldots, O_n\}$ of type T with initial object state s_0, and n additional atomic registers $\{R_1, R_2, \ldots, R_n\}$. The idea is that each process P_i "owns" one breakable object O_i and one register R_i. When P_i proposes v, it first writes v into R_i, and then invokes the *jumpAndCheck* method on O_i (line 4). Then it goes through objects $O_1, O_2, \ldots$, in this order, but skips its own object O_i, such that on each object O_j, it invokes the *saveAndCheck* method (line 6). From these invocations, it finds the smallest index b such that the invocation on object O_b (either *saveAndCheck* or *jumpAndCheck*) returns symbol BROKEN. It does not need to go further after finding such a b, and just returns the value stored in R_b as the consensus decision.

Theorem 2. *For a breakable object type T, if the methods jumpAndCheck and saveAndCheck together with an initial state s_0 satisfy the n-Break Condition, then Algorithm 1 implements n-process consensus.*

Algorithm 1: Consensus algorithm for n processes $\{P_1, P_2, \ldots, P_n\}$ using *jumpAndCheck* and *saveAndCheck* on n breakable objects $\{O_1, O_2, \ldots, O_n\}$ of type T with initial object state s_0, and n atomic registers $\{R_1, R_2, \ldots, R_n\}$. The code is for process P_i. Note that *jumpAndCheck* and *saveAndCheck* may not be atomic.

```
1 Initially, all objects O_1, O_2, ..., O_n have state s_0.
2 consensus(v) begin
3     write(R_i, v)
4     if jumpAndCheck(O_i) = BROKEN then b := i else b := n
5     foreach j ∈ {1, 2, ..., b}, j ≠ i, in ascending order do
6         if saveAndCheck(O_j) = BROKEN then
7             b := j
8             break
9     return read(R_b)
```

Proof. The *Termination* property is straightforward, because the loop is simple and finite.

In Algorithm 1, every O_i $(1 \leqslant i \leqslant n)$ is initially in state s_0. There are at most one invocation of method *jumpAndCheck* (by P_i) and at most $n-1$ invocations of method *saveAndCheck* (by processes other than P_i) on O_i. By the n-Break Condition, all these methods on O_i return the same symbol, and they return BROKEN if and only if *jumpAndCheck* is invoked first.

Every process invokes one *jumpAndCheck* before any *saveAndCheck* in Algorithm 1. For the process that first invokes *jumpAndCheck*, say P_j $(1 \leqslant j \leqslant n)$, one can see that the first method invocation on O_j is *jumpAndCheck*. Let $B \subseteq \{1, 2, \ldots, n\}$ be such that for every $b \in B$, the first method invocation on O_b is *jumpAndCheck*, and thus all method invocations on O_b returns BROKEN according to the n-Break Condition. B is not empty since $j \in B$.

By Algorithm 1, the final value of variable b in every process must be in B. For such a $b \in B$, the first invocation of method on O_b is *jumpAndCheck* by process P_b. Before this, P_b must have already written its proposal into R_b. Therefore the *Validity* property is satisfied.

Let b_0 be the smallest number in B. We claim that the final value of variable b in every process equals to b_0, which means they all decide on the value of R_{b_0}. If this is not true, assume that a process finds a number $b \neq b_0$. There must be a method *jumpAndCheck* or *saveAndCheck* on O_b that returns BROKEN. Hence all methods on O_b return BROKEN, and the first method invocation on O_b is *jumpAndCheck*. Therefore $b \in B$ and $b_0 < b$. By Algorithm 1, this process must have invoked *saveAndCheck* on O_{b_0} and that method returns SAFE. This is in contradiction with $b_0 \in B$. Therefore all processes decide on R_{b_0}, the *Agreement* property is satisfied.

In summary Algorithm 1 implements n-process consensus. □

[2] The framework can be easily generalized to the case of using multiple base objects to implement *jumpAndCheck* and *saveAndCheck* methods, but it is not needed in this paper.

In our general framework, methods *jumpAndCheck* and *saveAndCheck* may be implemented by multiple operations and thus may not be atomic. The following corollary covers the special case when indeed we can use the atomic jump and save operations of the object to implement *jumpAndCheck* and *saveAndCheck*.

Corollary 1. *Suppose a breakable object type T satisfies the following conditions*:

(a) *it has a cliff state c with a jump operation ω_J on c that returns $\perp_B$*;
(b) *it has a save operation ω_S on c*;
(c) *for any operation sequence consisting of at most $n-1$ ω_S and at most one ω_J, as long as the first operation is not ω_J, the sequence applied to state c will not enter the broken state.*

Then the consensus number of T is at least n. In particular, if ω_S always transitions a non-broken object state into a safe state, then the consensus number of T is infinity.

Proof. Let $s_0 = c$, BROKEN $= \perp_B$, *jumpAndCheck* be ω_J, *saveAndCheck* be ω_S, and consider every return value other than $\perp_B$ as the SAFE symbol. One can see that the n-Break Condition is satisfied. By the above theorem, Algorithm 1 implements n-process consensus.

Moreover, when ω_S always transfers a non-broken object state into a safe state, any one ω_J operation inserted in the middle of a sequence of ω_S's of any length will not break the object, so condition (c) is satisfied for any $n > 1$. Thus in this case the consensus number of T is infinity. □

The above corollary demonstrates that a cyclic breakable object could bring strong consensus power all the way to infinity, and its power is mainly due to its particular state transition pattern between the cliff, safe, and broken states, not on the values stored in the objects or the semantics of the operations. In the next two sections, we apply our framework to various breakable queues. The results are easily carried over to corresponding breakable stacks.

4. Object-wide breakable objects

In this section, we focus on object-wide breakable objects. In particular, we use various versions of breakable queues as our running examples to show how we apply our general algorithmic framework and Theorem 2 to provide lower bounds on their consensus numbers. We also provide matching upper bounds on the consensus numbers to show that our results are tight. We will point out that the actual semantics of the queue type is not important, and our results can be applied to other breakable objects.

Given a normal queue with *deq* and $enq(v)$ operations, we can introduce the broken state in several ways. First, we can revise the $enq(v)$ operation such that the queue has at most n cells and $enq(v)$ may break the full queue. Overall, we can introduce three types of enqueue operations: (a) normal, in which the queue has no limit and $enq(v)$ always succeeds with no return value; (b) n-cell normal, in which the queue has at most n cells, and when $enq(v)$ is applied to a full queue, the queue state does not change and $enq(v)$ has no return value as if it is a normal operation; and (c) n-cell break-all, in which the queue has at most n cells and when $enq(v)$ is applied to a full queue, the queue enters the broken state and $enq(v)$ returns $\perp_B$. The n-bounded queue defined in Section 2 is a queue with the n-cell break-all $enq(v)$ operations and a normal *deq* operation. Similarly, we can also revise the *deq* operation such that *deq* has the following two types: (a) normal, which is the normal *deq* operation; and (b) break-all, in which *deq* on an empty queue will transfer the queue into the broken state and return $\perp_B$. Combining three types of $enq(v)$ operations and two types of *deq* operations, we obtain six variants of queues. Their consensus numbers are fully known, and are listed in Table 1 (ignoring the last row and column for now).

All of our algorithmic results are based on our algorithmic framework, but different variants of queues require different realization of the two methods *jumpAndCheck* and *saveAndCheck*. The matching impossibility proofs all start by applying Lemma 1, but need different treatments based on the semantics of different objects. We now provide individual results below on different breakable objects, with a discussion on generalizing the results to other objects.

4.1. n-BQ: n-cell break-all $enq(v)$'s and normal *deq*

We start with the n-bounded queue, which contains the core operations as a variant defined in [6] and another variant defined in [2], Exercise 15.8. Contrary to what has been claimed in the literature, which imply that the consensus number of n-BQ is n, we show here that the consensus number of n-BQ is infinity. Algorithm 2 provides the implementation of our general framework using the n-BQ object. We summarize it here for completeness, even though our theorem can be obtained directly from Corollary 1.

Theorem 3. *The implementation in Algorithm 2 using the n-BQ object solves m-process consensus for all $n \geqslant 1$ and $m \geqslant 1$. Thus, the consensus number of n-bounded queue is infinity for any $n \geqslant 1$.*

Algorithm 2: Implementation of the general framework with n-BQ: n-cell break-all $enq(v)$'s and normal deq, $n \geqslant 1$.

```
1 initial State s_0: any full queue state
2 jumpAndCheck: if enq(v) = ⊥_B then return BROKEN else return SAFE
3 saveAndCheck: if deq = ⊥_B then return BROKEN else return SAFE
```

Proof. This is a direct application of Corollary 1. For any n-bounded queue with $n \geqslant 1$, pick any full queue state c as its cliff state, any $enq(v)$ as its jump operation on c, the deq operation is a save operation on c. The deq operation always transfers any non-broken queue state into a safe state (i.e., the queue will not be full after deq as long as it is not broken yet). Therefore, by Corollary 1, n-bounded queue has consensus number infinity. □

Even though the result seems to be easy after we have developed our generic algorithmic framework, the result is still important in providing several insights in our understanding of the consensus power of breakable objects.

First, the n-bounded queue achieves infinite consensus number without using the peek operation. This is in contrast to the normal queue and the normal augmented queue with the peek operation. The normal queue with normal $enq(v)$'s and deq only achieves consensus number 2. But after adding the peek operation, it achieves consensus number infinity. However, by just adding the ability of breaking a normal queue, and even if the queue can only have one cell, we are able to achieve consensus number infinity. This shows that consensus power can be greatly increased when an object may break.

Second, while n-BQ achieves infinite consensus number without using the peek operation, it actually does not use the content of the queue at all. All it needs is the state transition pattern in which enqueue on a full queue (the cliff state) can break the queue while a dequeue on a full queue (or any normal queue state) will bring the queue away from the cliff state. The same result can be applied to different breakable objects with the same state transition pattern. For example, consider a simple n-bounded counter object, which stores a natural number and has two operations *inc* and *dec*. *inc* operation will increment the counter value by one if the current value is less than n, and will break the counter if it is n. And *dec* operation will decrement the counter by one if the current value is greater than 0, keeps it 0 if it is already 0. Both *dec* and *inc* return the value of the counter after the change. Even though the semantics of the n-bounded counter is different from that of the n-bounded queue, it has the same state transition pattern, and thus we can also apply Corollary 1 to show that n-bounded counter has consensus number infinity. Therefore, our approach not only works on queues, but works on a general class of breakable objects.

Third, the deq operation makes a huge difference on the consensus power of breakable queues. If we remove the deq operation and add the *peek* operation, making it an n-bounded peek queue as in [6], then it is acyclic and the consensus number is reduced to n as shown in [6]. Note that [6] made an incorrect claim (after their Claim 5) saying that adding deq operation to the n-bounded peek queue will not change the consensus number of the resulting object. In contrast, we show here that adding deq will boost the consensus number to infinity. In general, deq represents save operations. It demonstrates that the availability of a save operation can greatly increase the consensus power of the breakable object, while the absence of it may greatly reduce the consensus power.

4.2. Breakable queue with break-all deq and normal or n-cell normal or n-cell break-all enq(v)'s

We now look at breakable queues in which the dequeue operation will break the object when the queue is empty. The implementation of the framework is symmetric to the previous case when enqueue operations break the queue, and it works for several different cases based on whether the queue is bounded and whether the enqueue operations also break the queue (Algorithm 3).

Algorithm 3: Implementation of the general framework with object having break-all deq with normal or n-cell normal or n-cell break-all $enq(v)$'s, $n \geqslant 1$.

```
1 initial State s_0: empty queue state
2 jumpAndCheck: if deq = ⊥_B then return BROKEN else return SAFE
3 saveAndCheck: if enq(v) = ⊥_B then return BROKEN else return SAFE
```

Theorem 4. *The implementation in Algorithm 3 using the breakable queue with break-all deq and normal or n-cell normal enq(v)'s solves m-process consensus for all $n \geqslant 1$ and $m \geqslant 1$. Thus, the consensus number of these two breakable queue types are infinity for any $n \geqslant 1$.*

Proof. This is a direct application of Corollary 1, with the empty queue as the cliff state, deq as the jump operation, and $enq(v)$ as the save operation, which always transfers a non-broken state into a safe state (a non-empty queue). □

Even though we use the same implementations above for both normal $enq(v)$ operations and break-all $enq(v)$ operations, the implications to consensus number are different. The following two theorems show that if both $enq(v)$'s and deq can break the queue, the consensus number is no longer infinity but $n + 1$.

Theorem 5. *The implementation in Algorithm 3 using the breakable queue with break-all deq and n-cell break-all enq(v)'s solves* $(n+1)$*-process consensus, for any* $n \geqslant 1$.

Proof. This is also an application of Corollary 1, with the empty queue as the cliff state, *deq* as the jump operation, and *enq*(v) as the save operation. In this case, we can allow n consecutive *enq*(v)'s from the empty queue without breaking the queue, so by Corollary 1, we can solve $(n+1)$-process consensus. However, the difference from Theorem 4 is that *enq*(v) may transfer a non-broken state, in particular the full queue state, into a broken state. □

Theorem 6. *Breakable queue type with break-all deq and n-cell break-all enq(v)'s cannot solve* $(n+2)$*-process consensus, for any* $n \geqslant 1$.

Proof. Suppose, for the purpose of a contradiction, that the statement is not true. Then we can find a critical configuration as stated in Lemma 1. Let the processes be $P_1, P_2, \ldots, P_{n+2}$. Assume without loss of generality that P_1's next step leads to a 1-valent configuration and P_2's next step leads to a 0-valent configuration. We conduct the following case analysis to reach a contradiction.

Case 1. The next operations of all processes are *enq*$(\cdot)$'s. The following two sequences of steps lead to opposite univalent configurations. However, if we let P_{n+2} run solo from now on, P_{n+2} cannot distinguish these two configurations since in both cases P_{n+2} gets $\perp_B$ as its return value and the queue is broken after $n+2$ *enq* operations — a contradiction.

1. $P_1, P_2, P_3, \ldots, P_{n+2}$ take steps in the order $P_1, P_2, P_3, \ldots, P_{n+2}$.
2. $P_1, P_2, P_3, \ldots, P_{n+2}$ take steps in the order $P_2, P_1, P_3, \ldots, P_{n+2}$.

Case 2. The next operations are all *deq*'s. We again consider the above two step sequences, which also lead to opposite univalent configurations, in which the queue is broken and the return value of P_{n+2}'s *deq* is $\perp_B$ in both cases. Thus, if we let P_{n+2} run solo, it cannot distinguish the two runs — a contradiction.

Case 3. The next operations contain both *enq*$(\cdot)$'s and *deq*'s. Without loss of generality, we assume that P_1's next step is *enq*(v) and P_2's next step is *deq*. We further divide this case into several subcases.

- The queue is full and there is another process P_3 whose next step is *enq*. In this case, the following two step sequences lead to opposite univalent configurations but P_1 cannot distinguish them, because P_1's *enq* in both cases will break the queue and return $\perp_B$.
  1. P_1 takes a step (*enq*).
  2. P_1, P_2, P_3 take steps in the order P_2, P_3, P_1 (*deq*, *enq*, *enq*).
- The queue is full and the next operations of $P_2, P_3, \ldots, P_{n+2}$ are all *deq*'s. In this case, the following two step sequences lead to opposite univalent configurations but P_1 cannot distinguish them, because the queue is broken by P_1's *enq* in the first case while by the $n+1$ *deq*'s in the second case.
  1. P_1 takes a step (*enq*).
  2. $P_1, P_2, P_3, \ldots, P_{n+2}$ take steps in the order $P_2, P_3, \ldots, P_{n+2}, P_1$ (*deq*, …, *deq*, *enq*).
- The queue is empty. This is symmetric to the above two cases.
- The queue is neither full nor empty. The following two sequences lead to the same configuration, contradicting the fact that they should lead to opposite univalent configurations.
  1. P_1, P_2 take steps in the order P_1, P_2 (*enq*, *deq*).
  2. P_1, P_2 take steps in the order P_2, P_1 (*deq*, *enq*).

There is a contradiction in every case. Therefore the consensus number is at most $n+1$. □

Again we point out that the above results are not limited to breakable queues. We can adapt the n-bounded counter object so that its *dec* operation will break the counter when the counter value is 0. Then the above results on breakable queues can be transferred to such breakable counter objects.

5. Operation-wise breakable objects

We now turn to operation-wise breakable objects, in which some operations may break but not other operations. We still use breakable queues as running examples, and extend both *enq*$(\cdot)$ and *deq* operations to allow the following variants: (a) n-cell break-op *enq*$(\cdot)$: It behaves normally when the queue is not full; when the queue is full, it returns $\perp_B$ and all subsequent *enq*$(\cdot)$ operations return $\perp_B$, but keeps the queue state unchanged (*deq* can still work on the queue); and (b)

break-op *deq*: It behaves normally when the queue is not empty; when it is empty, it returns $\perp_B$ and breaks all subsequent *deq* operations but leave the queue state unchanged (*enq*(·) can still work on the queue as normal).

We are still able to apply the general framework in this case, but the actual implementations of the framework are more involved than that in the object-wide breakable objects case. Our results are summarized in the last row and column of Table 1. After obtaining the technical results, we make comparisons between objects with break-all or break-op operations. Intuitively we find that the difference in their consensus power lies in their ability of communicating about their break status among processes.

5.1. Breakable queue with break-op deq and normal or n-cell normal or n-cell break-op enq(v)'s

Algorithm 4: Implementation of the general framework with object having break-op *deq* and normal or n-cell normal or n-cell break-op *enq*(v)'s, $n \geqslant 1$.

```
1 initial State s_0: empty queue state
2 jumpAndCheck: if deq = ⊥_B then return BROKEN else return SAFE
3 saveAndCheck: enq(v); enq(v); if deq = ⊥_B then return BROKEN else return SAFE
```

Algorithm 4 shows the implementation of the framework using a breakable queue with break-op *deq* and several kinds of *enq*(v) operations. We use empty queue as the initial state s_0, and *deq* as the *jumpAndCheck* method. Different from Algorithm 3, we cannot simply use an *enq*(v) for *saveAndCheck*. The reason is that *enq*(v) will detect the breaking of the queue when the entire queue breaks, but it will not detect any difference when only *deq* breaks. Thus, we need to do two *enq*(v)'s first, and then use *deq* to detect if *deq* is broken. The reason of using two *enq*(v)'s is to leave an extra element in the queue so as to avoid having a later *jumpAndCheck* to break the queue.

Theorem 7. *The implementation in Algorithm 4 using the breakable queue with break-op deq and normal enq(v)'s solves m-process consensus for all $n \geqslant 1$ and $m \geqslant 1$. Thus, the consensus number of this breakable queue type is infinity for any $n \geqslant 1$.*

Proof. There is at most one *jumpAndCheck*, and at most $m-1$ *saveAndCheck*'s invoked on each queue in an m-process system. We need to check the n-Break Condition for the implementation in Algorithm 4. If the first invocation is *jumpAndCheck* on a queue, a *deq* is invoked on an empty queue, so the *deq* is broken and *jumpAndCheck* method returns BROKEN. For any subsequent *saveAndCheck* method invocation, it contains a *deq* operation that is broken, so it will return BROKEN. If the first invocation is *saveAndCheck*, the first operation on the queue is *enq*(v). By Algorithm 4, there is at most one *deq* in each method, and the *deq* is after two *enq*(v)'s in *saveAndCheck*. Thus in any partial run with $m-1$ *saveAndCheck*'s and one *jumpAndCheck*, as long as the first invocation is *saveAndCheck*, the number of *deq*'s is no more than the number of previous *enq*'s. Hence the *deq* operation can never get broken and all methods return SAFE. Therefore m-Break Condition is satisfied. By Theorem 2, m-process consensus can be solved by our algorithm. □

Theorem 8. *The implementation in Algorithm 4 using the breakable queue with break-op deq and n-cell normal enq(v)'s or n-cell break-op enq(v)'s solves n-process consensus for all $n \geqslant 1$.*

Proof. There is at most one *jumpAndCheck*, and at most $n-1$ *saveAndCheck*'s invoked on each queue in an n-process system. If the first invocation is *jumpAndCheck* on a queue, the *deq* operation is broken, and all methods return BROKEN. Otherwise, if the first invocation is *saveAndCheck*, the first operation on the queue is *enq*(v). Before the queue becomes full, the same argument as in Theorem 7 can show *deq* does not get broken. After the queue becomes full with n entries, since there can be at most n *deq* operations in the $n-1$ *saveAndCheck*'s and one *jumpAndCheck*, *deq* does not get broken either. Note that, we only prove that *deq* operation does not get broken in this case. It is possible that *enq* operation has already broken (if it is n-cell break-op *enq*), but it does not affect the *deq* operation. By our algorithm, all methods return SAFE. Therefore n-Break Condition is satisfied. By Theorem 2, n-process consensus can be solved by our algorithm. □

Theorem 9. *Breakable queue with break-op deq and n-cell normal enq(v)'s cannot solve $(n+1)$-process consensus, for any $n \geqslant 2$.*

Proof. Suppose, for a contradiction, that this is not true. Then we can find a critical configuration as stated in Lemma 1. Let the processes be $P_1, P_2, \ldots, P_{n+1}$. Assume without loss of generality that P_1's next operation leads to a 1-valent configuration, and P_2's next operation leads to a 0-valent configuration.

Case 1. The next operations of P_1 and P_2 are both *enq*'s. Consider the following two step sequences that lead to opposite univalent configurations.

1. P_1, P_2 take steps in the order P_1, P_2.
2. P_1, P_2 take steps in the order P_2, P_1.

After this, the only possible difference between the two runs is on at most two elements in the queue. Precisely, if the queue had at most $n-2$ elements at the critical configuration, the difference is only the order of the last two elements in the queue; if the queue had $n-1$ elements at the critical configuration, the difference is only on the last one element in the queue; and if the queue was full at the critical configuration, there is no difference.

- If the queue had at most $n-2$ elements, we schedule the processes using the same idea as in [4] and [2] (page 325) to reach a contradiction. Let a and b be the elements of P_1 and P_2 to enqueue respectively. The difference of the two runs is the order of a, b in the queue ((a, b) in the first run and (b, a) in the second run). If we let P_1 run solo, it will eventually decide differently in the two runs. Hence P_1's internal states of the two runs must become different at some point. And before this point, P_1 must be taking internal steps and applying operations on shared objects identically in the two runs. The only operation that can cause P_1's internal states to differ is a *deq* when $\{a, b\}$ are at the first two positions of the queue. P_1's *deq* operation will return a in the first run and b in the second run. We suspend P_1 right before this *deq* operation. Now the only difference between the two runs is still the order of a, b, but $\{a, b\}$ are at the first two positions of the queue. A similar argument shows that if we turn to let P_2 run solo, it will eventually *deq* the first element of the queue. Let P_2 run solo until it reaches this *deq*. Then let P_1 and P_2 do their *deq*'s in a row. After that, the only differences between the two runs are the internal states of P_1 and P_2. Another process P_3 (here we use the condition $n \geqslant 2$) will not be able to distinguish the two runs if P_1 and P_2 are both suspended.
- If the queue had $n-1$ elements, the difference of the two run is on only one element. This is because the second *enq* in each run has no effect. Let P_1 run solo, it will eventually *deq* this element due to the same reason as in the above subcase. However, after that, any other process will not be able to distinguish the two runs if we suspend P_1.
- If the queue was full, there is no difference since neither *enq* can succeed. Hence no process is able to distinguish the two runs.

Case 2. The operations are both *deq*'s. We also consider the above two sequences, which lead to opposite univalent configurations, in which the queue state is the same. Then another process P_3 cannot distinguish the two runs if it runs solo (here we use the condition $n \geqslant 2$).

Case 3. P_1's next step is *enq* and P_2's next step is *deq*. We assume that every process whose next step is *enq* leads to a 1-valent configuration, and every process whose next step is *deq* leads to a 0-valent configuration, since all the other cases are covered by Cases 1 and 2 above. We consider the following subcases.

- If the queue is full, the following two sequences lead to opposite univalent configurations but P_2 cannot distinguish them, because P_1's *enq* in the second sequence has no effect on a full queue.
  1. P_2 takes a step (*deq*).
  2. P_1, P_2 take steps in the order P_1, P_2 (*enq*, *deq*).
- If the queue is empty and there is another process P_3 whose next step is *deq*, the following two sequences lead to opposite univalent configurations, but P_2 cannot distinguish them, because in both cases the queue is empty and P_2 gets $\perp_B$ from its *deq*.
  1. P_2 takes a step (*deq*).
  2. P_1, P_2, P_3 take steps in the order P_1, P_3, P_2 (*enq*, *deq*, *deq*).
- If the queue is empty and the next operations of $P_3, \ldots, P_{n+1}$ are all *enq*'s (and they all lead to 1-valent configurations), then consider the following two step sequences that lead to opposite univalent configurations.
  1. $P_1, P_2, P_3, \ldots, P_{n+1}$ take steps in the order $P_2, P_1, P_3, \ldots, P_{n+1}$ (*deq*, *enq*, ..., *enq*).
  2. $P_1, P_3, \ldots, P_{n+1}$ take steps in the order $P_1, P_3, \ldots, P_{n+1}$ (*enq*, ..., *enq*).

  The only possible differences between the two runs are the internal states of P_2 and that *deq* is broken in the first run. Each of the other processes $P_1, P_3, \ldots, P_{n+1}$ has identical internal states in the two runs. We use the same idea as in Case 1 to schedule these n processes. Let P_1 run solo first. It will eventually take a *deq*, since it cannot gain any information from *enq* operations and *deq* is the only operation that can return differently in the two runs. We suspend P_1 before its first *deq* on the queue. Now even though P_1 may have done some operations that affected the shared objects, they were all identical in both runs. We then similarly let each of $P_3, \ldots, P_{n+1}$ run solo, and suspend before the first *deq*'s. At last, we let all these n processes do their *deq*'s in a row. Since the queue was filled with n entries, these n *deq*'s will empty the queue. Afterwards, in the second run let P_2 take its *deq* operation. This *deq* operation will break all subsequent *deq* operations and change P_2's internal state to be the same as in the first run. At this point, the process P_2 cannot distinguish the two runs with all the other processes suspended.
- If the queue is neither full nor empty, the following two sequences lead to the same configuration, contradicting the fact that they should lead to opposite univalent configurations.
  1. P_1, P_2 take steps in the order P_1, P_2 (*enq*, *deq*).
  2. P_1, P_2 take steps in the order P_2, P_1 (*deq*, *enq*).

Case 4. P_1's next step is *deq* and P_2's next step is *enq*. This is symmetric to Case 3.

There is a contradiction in every case. Therefore the consensus number is at most n. □

Comparing the above theorem with Theorem 4, we can see that break-op dequeue operations together with n-cell normal enqueues have less consensus power than break-all dequeues with n-cell normal enqueues. The intuition is that break-op dequeue cannot directly communicate its broken status to the enqueue operation, but break-all dequeue can do so. Thus to communicate this status, we have to use two enqueues and then a dequeue in Algorithm 4 for the enqueues to detect the broken dequeues. This works fine if the queue is unbounded (Theorem 7), but if the queue is n-cell bounded, communications about the broken status are constrained, and it can only support n processes for consensus.

The boundary case of Theorem 9, namely when $n = 1$, is that breakable queue with break-op *deq* and 1-cell normal *enq*(v)'s has consensus number 2. It solves 2-process consensus by the standard method of using one queue filled with one value, and whoever successfully dequeues the value wins the consensus. The proof that it cannot solve 3-process consensus follows exactly as the proof of Theorem 9.

Theorem 10. *Breakable queue with break-op deq and n-cell break-op enq(v)'s cannot solve $(n+1)$-process consensus, for any $n \geqslant 3$.*

Proof. Suppose, for a contradiction, that this is not true. Then we can find a critical configuration as stated in Lemma 1. Let the processes be $P_1, P_2, \ldots, P_{n+1}$. Assume without loss of generality that P_1's next operation leads to a 1-valent configuration, and P_2's next operation leads to a 0-valent configuration.

Case 1. The next operations of P_1 and P_2 are both *enq*'s. Consider the following two step sequences that lead to opposite univalent configurations.

1. P_1 and P_2 take steps in the order P_1, P_2.
2. P_1 and P_2 take steps in the order P_2, P_1.

This is similar to Case 1 of Theorem 9. The difference is that here *enq* returns $\perp_B$ in the case of failure while it does not return anything in Theorem 9. Note that there are always equal number of *enq*'s and *deq*'s taken in both runs. Hence the number of elements in the queue is always the same, and the *enq* operation must always have the same state (broken or not) in both runs. This also implies that no process can distinguish the two runs through *enq* operations.

- If the queue had at most $n-2$ elements at the critical configuration, both P_1 and P_2 can successfully *enq*. The difference is the order of two elements at the tail of the queue. We schedule them in the same way as in Theorem 9. Let P_1 run solo until it reaches a *deq* when those two elements are at the head of the queue. Then let P_2 run solo until the first *deq*. Next if we let them do the *deq*'s, another process P_3 will have no way to distinguish the two runs.
- If the queue had $n-1$ elements at the critical configuration, only one of P_1 and P_2 can successfully *enq*. This subcase is different from Theorem 9, because both P_1 and P_2 can distinguish the two runs by the return value of *enq*. But we can apply the idea to another process P_3. Let P_3 run solo, it has to *deq* out the element enqueued by either P_1 or P_2 to see the difference. After that, P_4 cannot distinguish the two runs (here we need $n \geqslant 3$).
- If the queue was full, no process can distinguish the two runs.

Case 2. The next operations of P_1 and P_2 are both *deq*'s. We also consider the above two sequences, which lead to opposite univalent configurations. One can see that another process P_3 cannot distinguish the two runs.

Case 3. P_1's next step is *enq* and P_2's next step is *deq*. We assume that every process whose next step is *enq* leads to a 1-valent configuration, and every process whose next step is *deq* leads to a 0-valent configuration, since all the other cases are covered by Cases 1 and 2 above.

- If the queue is full and there is another process P_3 whose next step is *enq*, consider the following two sequences that lead to opposite univalent configurations.
  1. P_1, P_2 take steps in the order P_1, P_2 (*enq*, *deq*).
  2. P_1, P_2, P_3 take steps in the order P_2, P_3, P_1 (*deq*, *enq*, *enq*).

  The queue in the first run has $n-1$ elements, which are the same as the first $n-1$ elements of the queue in the second run. In both cases, the *enq* operation is broken by P_1's *enq*. The only differences between the two runs are P_3's internal states and that the queue in the second run has one more element at its tail. We use the idea in Case 1 of Theorem 9 again. Each of $P_1, P_2, P_4, \ldots, P_{n+1}$ has to do at least one *deq* to see the difference if it runs solo, because *enq* is broken and always returns the same value $\perp_B$ in both runs. Let them run solo one by one until the first *deq*'s. Then let $P_2, P_4, \ldots, P_{n+1}$ do their *deq*'s in a row. Each of these $n-1$ processes gets the same entry in both runs. Afterwards, in the second run let P_1 take its *deq* operation. Now the queue is empty in both runs. The only differences between the two runs are the internal states of P_1 and P_3. Thus $P_2, P_4, \ldots, P_{n+1}$ cannot distinguish the two runs if P_1 and P_3 are suspended.

- If the queue is full and the next operations of $P_2, P_3, \ldots, P_{n+1}$ are all *deq*'s (and they all lead to 0-valent configurations), consider the following two sequences that lead to opposite univalent configurations.
  1. P_1, P_2 take steps in the order P_1, P_2 (enq, deq).
  2. P_2 takes a step (deq).

  The difference between the two runs is that *enq* is broken in the first run. We consider what P_2 does if it runs solo after its *deq* operation at the critical configuration.
  - P_2's first operation on the queue after the *deq* is *enq*. Let P_2 run until it finishes this *enq* in both runs, and then for the second run let P_1 take its *enq* operation. Now in both runs, *enq* is broken by P_1's *enq* operation. The elements of the queue in the first run are the same as the first $n-1$ elements of the queue in the second run. The only differences between the two runs are the internal states of P_2 and that the queue in the second run has one more element. This scenario is very similar to the previous case except that P_3 is replaced by P_2. The processes $P_1, P_3, P_4, \ldots, P_{n+1}$ each have to do at least one *deq* to see the difference. We can schedule them in the same way as in the previous case, and get a contradiction.
  - P_2's first operation on the queue after the *deq* is also *deq*. Let P_2 run until it finishes this second *deq*, and then let $P_3, P_4, \ldots, P_{n+1}$ take their *deq* operations. Now the queue is empty and *deq* is broken in both runs. The only differences between the two runs are the internal states of P_1 and that *enq* is broken in the first run. We use the trick similarly to the previous cases. Each of $P_2, P_3, \ldots, P_{n+1}$ has to do an *enq* to see the difference if it runs solo. Let them run solo one by one until their first *enq*'s are finished. These *enq*'s have no effect in the first run since *enq* is already broken, but they fill up the queue in the second run. Then for the second run let P_1 take its *enq* operation, which breaks the *enq*. Now in both runs P_1 has taken one *enq* operation that returned $\perp_B$, and both *enq* and *deq* are broken (even though the content of the queue is not the same). In this scenario, P_1 cannot distinguish the two runs.
- If the queue is empty and there is another process P_3 whose next step is *deq*, the following two sequences lead to opposite univalent configurations but P_2 cannot distinguish them.
  1. P_2 takes a step (deq).
  2. P_1, P_2, P_3 take steps in the order P_1, P_3, P_2 (enq, deq, deq).
- If the queue is empty and the next operations of $P_3, \ldots, P_{n+1}$ are all *enq*'s (and they all lead to 1-valent configurations), consider the following two runs that lead to opposite univalent configurations.
  1. P_1, P_2 take steps in the order P_2, P_1 (deq, enq).
  2. P_1 takes a step (enq).

  The difference between the two runs is that *deq* is broken in the first run. We consider what P_1 does if it runs solo after its *enq* operation at the critical configuration.
  - P_1's first operation on the queue after that *enq* is *deq*. In both runs, let P_1 run until it finishes this *deq*, and then in the second run let P_2 takes its *deq* operation. Now in both runs the queue is empty and *deq* is broken by P_2's *deq* operation. The process P_2 cannot distinguish the two runs.
  - P_1's first operation on the queue after that *enq* is also *enq*. In both runs, let P_1 run until it finishes this second *enq*, and then let $P_3, P_4, \ldots, P_{n+1}$ take their *enq* operations. Now *enq* is broken in both runs, and the only differences are the internal states of P_2 and that *deq* is broken in the first run. Each of $P_1, P_3, P_4, \ldots, P_{n+1}$ has to do an *deq* to see the difference if it runs solo. Let these n processes run solo one by one until their first *deq*'s are finished. Then for the second run let P_2 take its *deq* operation. Now in both runs P_2 has taken one *deq* operation that returned $\perp_B$, and both *enq* and *deq* are broken. In this scenario, P_2 cannot distinguish the two runs.
- If the queue is neither full nor empty. The following two sequences lead to the same configuration, contradicting the fact that they should lead to opposite univalent configurations.
  1. P_1, P_2 take steps in the order P_1, P_2 (enq, deq).
  2. P_1, P_2 take steps in the order P_2, P_1 (deq, enq).

Case 4. P_1's next step is *deq* and P_2's next step is *enq*. This is symmetric to Case 3.

There is a contradiction in every case. Therefore the consensus number is at most n. □

For boundary case of the above theorem when $n = 1$ or 2, their consensus numbers are all 3. The algorithm implementing 3-process consensus using these boundary objects are nontrivial. For completeness, the algorithm and the correctness proof are included in Appendix A.

5.2. Breakable queue with n-cell break-op enq(v)'s and normal deq

We now look at breakable queues where enqueue operations will break when the queue is full, but dequeue operation behaves normally. This is a case where we cannot treat enqueue and dequeue operations in a symmetric manner: its consensus number turns out to be $2n$, while a breakable queue with n-cell normal enqueues and break-op dequeue has consensus number n. The implementation of the framework is more involved, and it is the only case when we need to use two different values in the queue.

The idea starts by using a symmetric treatment of Algorithm 4: use a full queue as the initial state s_0, enqueue as *jumpAndCheck*, two dequeues followed by an enqueue as *saveAndCheck*. This satisfies n-Break Condition. When the number m of *saveAndCheck* invocations is between n and $2n-1$, it is possible that all m enqueues in *saveAndCheck* are scheduled together and thus breaking the enqueue operation even if *jumpAndCheck* is not invoked first. However, in this case, the queue is full and thus the content of the queue could leave a trace indicating that the enqueue is broken not because the *jumpAndCheck* is invoked first. This is the reason that in Algorithm 5, we initialize the full queue with all values x, and when doing enqueue, we always enqueue a different value y. Then even after the enqueue is broken, we invoke another *deq* to check if the queue contains y, and if so, the invocation still returns SAFE.

Algorithm 5: Implementation of the general framework with object having n-cell break-op $enq(v)$'s and normal *deq*. Let x and y be two values in the value domain of the object, $x \neq y$.

```
1 initial State s_0: full queue filled with value x
2 jumpAndCheck: if enq(y) ≠ ⊥_B then return SAFE else if deq = y then return SAFE else return BROKEN
3 saveAndCheck: begin
4     if deq = y then
5         return SAFE
6     else if deq = y then
7         return SAFE
8     else if enq(y) ≠ ⊥_B then return SAFE else if deq = y then return SAFE else return BROKEN
```

Theorem 11. *The implementation in Algorithm 5 using the breakable queue with n-cell break-op enq(v)'s and normal deq solves 2n-process consensus for all* $n \geqslant 1$.

Proof. We focus on one queue and prove the $2n$-Break Condition. There are at most one invocation of *jumpAndCheck* and at most $2n-1$ invocations of *saveAndCheck* on this queue.

If the first invocation is *jumpAndCheck*, the first operation is $enq(y)$. This makes the *enq* operation broken, and all *deq* operations return x or $\perp_E$ because no y can ever be put into the queue. Following Algorithm 5, both *jumpAndCheck* and *saveAndCheck* methods return BROKEN.

Otherwise, if the first invocation is *saveAndCheck*, the first operation is *deq*. Before the *enq* operation gets broken, no methods return BROKEN. This can be seen from the only *enq* operation in *jumpAndCheck* and *saveAndCheck*, which does not return $\perp_B$. In both methods, there is at most one *enq* operation. Moreover, in the *saveAndCheck* method, the *enq* operation is after two *deq*'s. One can see that the queue must have once been emptied before *enq* is broken. Thus if the queue is never emptied then all methods return SAFE. Suppose the queue becomes empty at some point. We consider the scenario right after the queue becomes empty. The first n *enq* operations do not return $\perp_B$ on the empty queue, which means the corresponding n method invocations return SAFE. These processes also fill the queue with value y. Thus if *enq* breaks, the queue must contain n y's at the time *enq* breaks. At the moment that the *enq* is broken, there are at most $n-1$ *saveAndCheck*'s and one *jumpAndCheck* that has not returned yet. According to Algorithm 5, each of these method invocations will invoke exactly one *deq* operation, which returns y, making the method return SAFE. Hence all method invocations return SAFE.

Therefore $2n$-Break Condition is satisfied. By Theorem 2, our algorithm solved $2n$-process consensus. □

Theorem 12. *Breakable queue with n-cell break-op enq(v)'s and normal deq cannot solve* $(2n+1)$*-process consensus, for any* $n \geqslant 2$.

Proof. Suppose, for a contradiction, that this is not true. We can find a critical configuration as stated in Lemma 1. Let the processes be $P_1, P_2, \ldots, P_{2n+1}$. Assume without loss of generality that P_1's next operation leads to a 1-valent configuration, and P_2's next operation leads to a 0-valent configuration.

Case 1. The next operations of P_1 and P_2 are both *enq*'s. Consider the following two runs that lead to opposite univalent configurations.

1. P_1, P_2 take steps in the order P_1, P_2.
2. P_1, P_2 take steps in the order P_2, P_1.

By the same argument as in Case 1 of Theorem 10, we can reach a contradiction if there are at least 4 processes (here we use the condition $n \geqslant 2$).

Case 2. The next operations of P_1 and P_2 are both *deq*'s. We also consider the above two runs, which lead to opposite univalent configurations. In this case, another process P_3 cannot distinguish the two runs.

Case 3. P_1's next step is *enq* and P_2's next step is *deq*. We assume that every process whose next step is *enq* leads to a 1-valent configuration, and every process whose next step is *deq* leads to a 0-valent configuration, since all the other cases are covered by Cases 1 and 2 above.

- The queue is full and there is another process P_3 whose next step is *enq*. This case can be argued verbatim as the first subcase of Case 3 in the proof of Theorem 10, so we do not repeat it here.
- The queue is full and the next operations of $P_2, P_3, \ldots, P_{2n+1}$ are all *deq*'s (and they all lead to 0-valent configurations). This is the key case that differs from the corresponding one of Theorem 9 and leads to the difference in the consensus power of the two cases. Consider the following two step sequences that lead to opposite univalent configurations.
  1. $P_1, P_2, P_3, \ldots, P_{n+1}$ take steps in the order $P_1, P_2, P_3, \ldots, P_{n+1}$ $(enq, deq, \ldots, deq)$.
  2. $P_2, P_3, \ldots, P_{n+1}$ take steps in the order $P_2, P_3, \ldots, P_{n+1}$ $(deq, \ldots, deq)$.

  The queue becomes empty after both runs. The only differences are the internal states of P_1 and that *enq* is broken in the first run. We use the same idea as in Case 1 of Theorem 9. Each of $P_2, P_3, \ldots, P_{n+1}$ has to do at least one *enq* if it runs solo, because *enq* is the only operation that returns differently in the two runs and the process should decide differently in the end. Let these n processes run solo one by one, and suspend right before the first *enq*'s. During this procedure *deq*'s may have been applied on the queue, but they do not affect the state because the queue is empty. Then we let the processes take the n *enq*'s in a row. In the first run the queue is still empty while in the second run the queue becomes full. Afterwards in the second run, let P_1 take its *enq*, which breaks the operation, and let $P_{n+2}, P_{n+3}, \ldots, P_{2n+1}$ take their *deq*'s to empty the queue. At this point in both runs the process P_1 has done one *enq* which returned $\perp_B$, and the queue has the same state. Hence the process P_1 cannot distinguish the two runs if it runs solo.
- If the queue is empty. The following two sequences lead to opposite univalent configurations but P_1 cannot distinguish them.
  1. P_1 takes a step (enq).
  2. P_2, P_1 take steps in the order P_2, P_1 (deq, enq).
- If the queue is neither full nor empty. The following two sequences lead to the same configuration, contradicting the fact that they should lead to opposite univalent configurations.
  1. P_1, P_2 take steps in the order P_1, P_2 (enq, deq).
  2. P_1, P_2 take steps in the order P_2, P_1 (deq, enq).

Case 4. P_1's next step is *deq* and P_2's next step is *enq*. This is symmetric to Case 3.

There is a contradiction in every case. Therefore the consensus number is at most $2n$. □

Comparing with the seemingly symmetric case of n-cell normal enqueues and break-op dequeue, the consensus number of breakable queues with n-cell break-op enqueues and normal dequeue is doubled. The intuitive reason is that we can use the difference between the initial and later content of the queue to convey the break status to more processes in the latter object type, but in the former type the initial state has to be empty, reducing the power of communicating about the break status.

For the boundary case of the above theorem when $n = 1$, again its consensus number is 3, and can be shown using the same algorithm included in Appendix A.

5.3. Breakable queue with n-cell break-all enq(v)'s and break-op deq

In this section, we show that the consensus number of breakable queue with n-cell break-all $enq(v)$'s and break-op *deq* is $2n$.

Algorithm 6: Implementation of the general framework with object having n-cell break-all $enq(v)$'s and break-op *deq*.

```
1 initial State s0: any full queue state
2 jumpAndCheck: if enq(v) = ⊥B then return BROKEN else return SAFE
3 saveAndCheck: if deq ≠ ⊥B then return SAFE else if enq(v) = ⊥B then return BROKEN else return SAFE
```

Theorem 13. *The implementation in Algorithm 6 using the breakable queue with n-cell break-all enq(v)'s and break-op deq solves 2n-process consensus for all* $n \geqslant 1$.

Proof. There is at most one *jumpAndCheck*, and at most $2n-1$ *saveAndCheck* invoked on each queue in a $2n$-process system. If the first invocation is *jumpAndCheck* on a queue, the queue goes into the broken state, and all methods return BROKEN. Otherwise, if the first invocation is *saveAndCheck*, the first operation on the queue is *deq*. By the algorithm, before *deq* becomes broken no *saveAndCheck* takes any $enq(v)$ operation, so they do not return BROKEN. The only way that *deq* becomes broken (but the queue is not broken) is for at least n processes to invoke *saveAndCheck* method and successfully *deq*

(not return $\perp_B$). These n invocations of *saveAndCheck* return SAFE. After *deq* becomes broken, the queue is empty. The remaining (at most n) processes' methods return SAFE because their calls to $enq(v)$ do not break the queue. Therefore $2n$-Break Condition is satisfied. By Theorem 2, $2n$-process consensus can be solved by our algorithm. □

Theorem 14. *Breakable queue with n-cell break-all $enq(v)$'s and break-op deq cannot solve $(2n+1)$-process consensus, for any $n \geqslant 1$.*

Proof. Suppose, for a contradiction, that this is not true. We can find a critical configuration as stated in Lemma 1. Let the processes be $P_1, P_2, \ldots, P_{2n+1}$. Assume without loss of generality that P_1's next operation leads to a 1-valent configuration, and P_2's next operation leads to a 0-valent configuration.

Case 1. The next operations of P_1 and P_2 are both *enq*'s. Consider the following two step sequences that lead to opposite univalent configurations.

1. P_1, P_2 take steps in the order P_1, P_2.
2. P_1, P_2 take steps in the order P_2, P_1.

If one of P_1 and P_2 failed to *enq* (return $\perp_B$), P_3 cannot distinguish the two runs because the queue is broken. If both P_1 and P_2 successfully finished *enq*, the difference between the two runs would be the order of the last two elements in the queue. We can reach a contradiction by the same argument as in Case 1 of Theorem 9.

Case 2. The operations are both *deq*'s. We also consider the above two runs, which lead to opposite univalent configurations. One can see that another process P_3 cannot distinguish the two runs.

Case 3. P_1's next step is *enq* and P_2's next step is *deq*. We assume that every process whose next step is *enq* leads to 1-valent, and every process whose next step is *deq* leads to 0-valent, since all the other cases are covered by Cases 1 and 2 above.

- The queue is full and there is another process P_3 whose next step is *enq*. The following two sequences lead to opposite univalent configurations but P_1 cannot distinguish them.
  1. P_1 takes a step (*enq*).
  2. P_1, P_2, P_3 take steps in the order P_2, P_3, P_1 (*deq*, *enq*, *enq*).
- The queue is full and the next operations of $P_3, P_4, \ldots, P_{2n+1}$ are all *deq*'s (and they all lead to 0-valent configurations). This is the case where we need $2n+1$ processes. Consider the following two sequences that lead to opposite univalent configurations.
  1. $P_1, P_2, P_3, \ldots, P_{2n+1}$ take steps in the order $P_1, P_2, P_3, \ldots, P_{2n+1}$ (*enq*, *deq*, ..., *deq*).
  2. $P_1, P_2, P_3, \ldots, P_{2n+1}$ take steps in the order $P_2, P_3, \ldots, P_{2n+1}, P_1$ (*deq*, ..., *deq*, *enq*).

  The queue is full in the first run, while it has one element in the second run. In both runs, *deq* is broken and the return values for $P_{n+2}, \ldots, P_{2n+1}$ are all $\perp_B$. The difference is that *enq* is broken in the first run. We use the idea in Case 1 of Theorem 9 again. Each of $P_{n+2}, \ldots, P_{2n+1}$ has to do an *enq* if it runs solo, because *enq* is the only operation that returns differently in the two runs. Let these n processes run solo one by one until their first *enq*'s are finished. In the second run the queue is broken by P_{2n+1}'s *enq*. In both cases P_{2n+1} gets the return value $\perp_B$ and the queue is broken. Thus the process P_{2n+1} cannot distinguish the two runs.
- If the queue is empty and there is another process P_3 whose next step is *deq*, the following two sequences lead to opposite univalent configurations but P_2 cannot distinguish them.
  1. P_2 takes a step (*deq*).
  2. P_1, P_2, P_3 take steps in the order P_1, P_3, P_2 (*enq*, *deq*, *deq*).
- If the queue is empty and the next operations of $P_3, P_4, \ldots, P_{2n+1}$ are all *enq*'s (and they all lead to 1-valent configurations). The following two sequences lead to opposite univalent configurations but P_{n+2} cannot distinguish them since the queue is broken in both runs.
  1. $P_1, P_2, \ldots, P_{n+2}$ take steps in the order $P_2, P_1, P_3 \ldots, P_{n+2}$ (*deq*, *enq*, ..., *enq*).
  2. $P_1, P_3, \ldots, P_{n+2}$ take steps in the order $P_1, P_3, \ldots, P_{n+2}$ (*enq*, ..., *enq*).
- If the queue is neither full nor empty. The following two sequences lead to the same configuration, contradicting the fact that they should lead to opposite univalent configurations.
  1. P_1, P_2 take steps in the order P_1, P_2 (*enq*, *deq*).
  2. P_1, P_2 take steps in the order P_2, P_1 (*deq*, *enq*).

Case 4. P_1's next step is *deq* and P_2's next step is *enq*. This is symmetric to Case 3.

There is a contradiction in every case. Therefore the consensus number is at most $2n$. □

5.4. Breakable queue with n-cell break-op enq(v)'s and break-all deq

In this section, we show that the consensus number of breakable queue with n-cell break-op $enq(v)$'s and break-all deq is $2n$. The technique is similar to the previous section.

Algorithm 7: Implementation of the general framework with object having n-cell break-op $enq(v)$'s and break-all deq.

1 initial State s_0**:** empty queue
2 *jumpAndCheck*: **if** $deq = \perp_B$ **then return** BROKEN **else return** SAFE
3 *saveAndCheck*: **if** $enq(v) \neq \perp_B$ **then return** SAFE **else if** $deq = \perp_B$ **then return** BROKEN **else return** SAFE

Theorem 15. *The implementation in Algorithm 7 using the breakable queue with n-cell break-op enq(v)'s and break-all deq solves $2n$-process consensus for all $n \geqslant 1$.*

Proof. There is at most one *jumpAndCheck*, and at most $2n-1$ *saveAndCheck* invoked on each queue in a $2n$-process system. If the first invocation is *jumpAndCheck* on a queue, the queue goes into the broken state, and all methods return BROKEN. Otherwise, if the first invocation is *saveAndCheck*, the first operation on the queue is $enq(v)$. By the algorithm, before enq becomes broken no *saveAndCheck* takes a deq operation, so they do not return BROKEN. The only way that enq becomes broken (but the queue is not broken) is for at least n processes to invoke *saveAndCheck* method and successfully enq (not return $\perp_B$). These n invocations of *saveAndCheck* return SAFE. After enq becomes broken, the queue is full. The remaining (at most n) processes' methods return SAFE because all their deq's are successful. Therefore $2n$-Break Condition is satisfied. By Theorem 2, $2n$-process consensus can be solved by our algorithm. □

Theorem 16. *Breakable queue with n-cell break-op enq(v)'s and break-all deq cannot solve $(2n+1)$-process consensus, for any $n \geqslant 2$.*

Proof. Suppose, for a contradiction, that this is not true. We can find a critical configuration as stated in Lemma 1. Let the processes be $P_1, P_2, \ldots, P_{2n+1}$. Assume without loss of generality that P_1's next operation leads to a 1-valent configuration, and P_2's next operation leads to a 0-valent configuration.

Case 1. The next operations of P_1 and P_2 are both enq's. Consider the following two step sequences that lead to opposite univalent configurations.

1. P_1, P_2 take steps in the order P_1, P_2.
2. P_1, P_2 take steps in the order P_2, P_1.

By the same argument as in Case 1 of Theorem 10, we can reach a contradiction if there are at least 4 processes (here we use the condition $n \geqslant 2$).

Case 2. The next operations of P_1 and P_2 are both deq's. We also consider the above two sequences, which lead to opposite univalent configurations. One can see that another process P_3 cannot distinguish the two runs.

Case 3. P_1's next step is enq and P_2's next step is deq. We assume that every process whose next step is enq leads to a 1-valent configuration, and every process whose next step is deq leads to a 0-valent configuration, since all the other cases are covered by Cases 1 and 2 above.

- The queue is full and there is another process P_3 whose next step is enq. This case can be argued verbatim as the first subcase of Case 3 in the proof of Theorem 10, so we do not repeat it here.
- The queue is full and the next operations of $P_3, P_4, \ldots, P_{2n+1}$ are all deq's (and they all lead to 0-valent configurations). The following two sequences lead to opposite univalent configurations but P_{n+2} cannot distinguish them since the queue is broken in both runs.
  1. $P_1, P_2, P_3, \ldots, P_{n+2}$ take steps in the order $P_1, P_2, P_3, \ldots, P_{n+2}$ $(enq, deq, \ldots, deq)$.
  2. $P_2, P_3, \ldots, P_{n+2}$ take steps in the order $P_2, P_3, \ldots, P_{n+2}$ $(deq, \ldots, deq)$.
- The queue is empty and there is another process P_3 whose next step is deq. The following two sequences lead to opposite univalent configurations but P_2 cannot distinguish them.
  1. P_2 takes a step (deq).
  2. P_1, P_2, P_3 take steps in the order P_1, P_3, P_2 (enq, deq, deq).
- The queue is empty and the next operations of $P_3, P_4, \ldots, P_{2n+1}$ are all enq's (and they all lead to 1-valent configurations). This is the case where we need $2n+1$ processes. Consider the following two sequences that lead to opposite univalent configurations.
  1. $P_1, P_2, P_3, \ldots, P_{2n+1}$ take steps in the order $P_2, P_1, P_3, \ldots, P_{2n+1}$ $(deq, enq, \ldots, enq)$.

2. $P_1, P_2, P_3, \ldots, P_{2n+1}$ take steps in the order $P_1, P_3, \ldots, P_{2n+1}, P_2$ ($enq, \ldots, enq, deq$).
The queue is empty in the first run, while it has $n-1$ elements in the second run. In both runs, *enq* is broken and the return values for $P_{n+2}, \ldots, P_{2n+1}$ are all $\perp_B$. The difference is that *deq* is broken in the first run. We use the idea in Case 1 of Theorem 9 again. Each of $P_{n+2}, \ldots, P_{2n+1}$ has to do a *deq* if it runs solo, because *deq* is the only operation that returns differently in the two runs. Let these n processes run solo one by one until their first *deq*'s are finished. In the second run the queue is broken by P_{2n+1}'s *deq*. In both cases P_{2n+1} gets the return value $\perp_B$ and the queue is broken. Thus the process P_{2n+1} cannot distinguish the two runs.

– If the queue is neither full nor empty. The following two sequences lead to the same configuration, contradicting the fact that they should lead to opposite univalent configurations.
1. P_1, P_2 take steps in the order P_1, P_2 (enq, deq).
2. P_1, P_2 take steps in the order P_2, P_1 (deq, enq).

Case 4. P_1's next step is *deq* and P_2's next step is *enq*. This is symmetric to Case 3.

There is a contradiction in every case. Therefore the consensus number is at most $2n$. □

For the boundary case of the above theorem when $n = 1$, the same algorithm in the appendix shows that its consensus number is 3.

Comparing the result in the above two sub-sections with results on breakable queues with both break-all *deq* and n-cell break-all *enq*'s (Theorems 5 and 6), queues with one-end break-all operations have higher consensus power than queues with two-end break-all operations. We can see the reason is that two-end break-all operations are too rigid, making it harder to communicate about the break status; while for one-end break-all operations with only break-op operation at the other end, even if the other operation breaks, the queue is still working and can still be used to communicate about the break status, and thus it has higher consensus power.

Finally, we point out that all the algorithmic results in this and the previous section apply to corresponding breakable stack objects, and thus our results are not specific to breakable queues only.

6. n-Consensus object with no broken states

To the best of our knowledge, so far all general objects with consensus number n are breakable objects, e.g. n-bounded peek queue of [6], n-bounded compareAndSet of [5], and the objects we studied in Table 1. In [4], Herlihy defines an n-register assignment object, which is non-breakable but has consensus number $2n$. Naturally, one may ask what would be a natural non-breakable object with consensus number n. Of course, n-consensus object itself has consensus number n, but we do not consider it as "natural" since (a) it is an artificial object used to define consensus number, (b) it is one-shot, meaning that it could only be used to decide one value and once the first value is locked in, its state will not change (in this sense, it is similar to breakable objects), and (c) it limits its power by restricting the number of ports for process to access.

In this section, we provide a variant of augmented queue object and show that its consensus number is exactly n, for any positive integer $n \geqslant 2$. It is very similar to n-bounded augmented queue, with normal *deq* and *peek* operations, and the $enq(v)$ operation that behaves normally when the queue is not full. When the queue is full, the next $enq(v)$ operation will insert v to the end of queue and remove the head of queue, so that the queue still has n entries after the operation, and it has no special return value. This $enq(v)$ operation at full queue acts as shifting the queue, so we call it *n-bounded shift augmented queue* (n-BSAQ). We show below that n-BSAQ has consensus number exactly n, for any $n \geqslant 2$.

Theorem 17. *For any $n \geqslant 2$, n-bounded shift augmented queue has consensus number n.*

The theorem is proven by the following two lemmas.

Lemma 2. *The n-bounded shift augmented queue can solve n-process consensus.*

Proof. We show that the following simple algorithm can implement n-process consensus. Let the processes be $P_1, P_2, \ldots, P_n$, and P_i's proposal is v_i ($1 \leqslant i \leqslant n$). All processes use one n-bounded shift augmented queue Q, which is initially empty. Every process P_i first $enq(v_i)$, then decide on the return value of *peek* on Q.

The *Termination* property is straightforward. The *Validity* property is because every element put into Q must be the proposal of some process. For the *Agreement* property, one can see that all processes decide on the first element ever put into Q, because there are n processes and no element is removed out of the queue.

Therefore the n-bounded shift augmented queue can solve n-process consensus. □

Lemma 3. *The n-bounded shift augmented queue cannot solve $(n+1)$-process consensus for any $n \geqslant 2$.*

Proof. Suppose, for a contradiction, that this is not true. Then we can find a critical configuration as stated in Lemma 1. Let the processes be $P_1, P_2, \ldots, P_{n+1}$. The next operation of every process is not *peek* because *peek* is trivial. Assume without loss of generality that P_1's next operation leads to a 1-valent configuration, and P_2's next operation leads to a 0-valent configuration.

Case 1. The next operations of P_1 and P_2 are both *deq*'s. The following two step sequences lead to opposite univalent configurations but P_3 cannot distinguish them (here we use the condition $n \geqslant 2$).

1. P_1, P_2 take steps in the order P_1, P_2.
2. P_1, P_2 take steps in the order P_2, P_1.

Case 2. P_1's next operation differs from P_2's. Without loss of generality, say P_1's next operation is *enq* and P_2's next operation is *deq*.

- The queue is empty. The following two sequences lead to opposite univalent configurations but P_1 cannot distinguish them.
  1. P_1 takes a step (*enq*).
  2. P_1, P_2 take steps in the order P_2, P_1 (*deq*, *enq*).
- The queue is full. We also consider the above two sequences, which lead to opposite univalent configurations. Since the *enq* of P_1 in the first run will shift the full queue, it is equivalent as a dequeue and then an enqueue, thus process P_1 cannot distinguish the two runs.
- The queue is neither full nor empty. The following two sequences lead to the same configuration, contradicting the fact that they should lead to opposite univalent configurations.
  1. P_1, P_2 take steps in the order P_1, P_2 (*enq*, *deq*).
  2. P_1, P_2 take steps in the order P_2, P_1 (*deq*, *enq*).

Case 3. Every process' next operation is *enq*. The following two sequences lead to opposite univalent configurations, but P_{n+1} cannot distinguish them, because P_{n+1}'s *enq* will remove the entry enqueued by P_1 in the first run, making it the same as in the second run where P_1 did not do an *enq*.

1. $P_1, P_2, P_3, \ldots, P_{n+1}$ take steps in the order $P_1, P_2, P_3, \ldots, P_{n+1}$.
2. $P_2, P_3, \ldots, P_{n+1}$ take steps in the order $P_2, P_3, \ldots, P_{n+1}$.

There is a contradiction in every case. Therefore the consensus number is at most n. □

7. Concluding remarks

In this paper, we provide a systematic study of breakable objects, focusing on cyclic breakable objects that are not well covered before. For breakable queues, we also considered several other variants, such as adding the peek operation, allowing jump operations to return a symbol different from $\perp_B$, allowing n-cell normal enqueue operation to return a special symbol when operating on a full queue, etc. We are still able to use our generic framework developed to implement consensus and find matching impossibility results, but we omit their presentation in the paper.

One open problem on breakable objects is to find the minimum number of objects needed to implement consensus. Currently, our framework needs n objects to implement n-process consensus. What is the minimum number of objects (say n-bounded queues) to solve n-process consensus? Another direction is to look into other general conditions on breakable objects either allowing solving consensus or showing impossibility results.

Appendix A. Boundary cases for breakable queues with n-cell break-op *enq*$(\cdot)$ operations, when $n = 1$ or 2

Theorem 18. *The consensus number of breakable queues with* 1*-cell break-op* enq$(\cdot)$ *operations and normal or break-all or break-op deq operation is* 3.

Proof. We first show that Algorithm 8 implements 3-process consensus. The *Termination* property is straightforward. There is one and only one process can successfully enqueue in line 6. Without loss of generality, say P_1 succeeds on line 6 and returns on line 7, which means it decides on its own proposal. The other two processes P_2 and P_3 must go to line 9, before which P_1 already saved its proposal in R_1. We just need to show that both P_2 and P_3 return the proposal of P_1, so that the *Validity* and *Agreement* properties are satisfied. P_2 and P_3 try *deq*(Q) on line 10, and exactly one gets the value enqueued by P_1. Say P_2 gets it, we can see P_2 returns the proposal of P_1 on line 13. For the process P_3, its *deq*(Q) operation either returns $\perp_E$ (for a normal *deq* operation) or $\perp_B$ (for a break-op or break-all *deq* operation), and thus it must execute line 12.

Algorithm 8: Consensus algorithm for 3 processes $\{P_1, P_2, P_3\}$ using breakable queues with 1-cell break-op $enq(\cdot)$ operations and normal or break-all or break-op deq. The code is for process P_i.

```
1  Initialization: begin
2      A breakable queue Q, empty
3      Registers R1, R2, R3, all with ⊥
4  consensus(v) begin
5      write(Ri, v)
6      if enq(Q, v) ≠ ⊥B then
7          return v
8      else
9          write(Ri, ⊥)
10         x = deq(Q)
11         if x = ⊥E or ⊥B then
12             return the non-⊥ value in read(R1), read(R2), read(R3)
13         else return x
```

Before P_3 executes line 12, R_2 and R_3 were cleaned to $\perp$ at line 9. So R_1 is the only register with non-$\perp$, and P_3 also returns the proposal of P_1.

By the same case study and arguments in Theorems 16, 12 and 10 respectively, we can show that the 3 types of 1-cell queues cannot implement 4-process consensus. Therefore the consensus number of these queues is 3. (Some cases degenerate for 1-cell queue, e.g. the case that the queue is neither empty nor full no longer exists.) □

Theorem 19. *The consensus number of breakable queues with* 2*-cell break-op* $enq(\cdot)$ *operations and break-op deq operation is* 3.

Proof. We can slightly modify Algorithm 8 as follows to show that it implements 3-process consensus when the breakable queue has 2-cell break-op operations. First, we initially insert an entry $\perp$ into the queue. Second, at line 10, we replace $deq(Q)$ with two $deq(Q)$'s. Let $x = \perp_E$ if neither $deq(Q)$ returns a valid proposal (a non-$\perp$ symbol), and let x be that proposal otherwise. The modified algorithm implements 3-process consensus using a breakable queue with 2-cell break-op $enq(\cdot)$ operations.

By the same case study and arguments in Theorem 10, we can show this queue cannot implement 4-process consensus. Therefore the consensus number of breakable queues with 2-cell break-op $enq(\cdot)$ and break-op deq is 3. □

References

[1] Y. Afek, E. Shalom, Consensus gaps between restricted and unrestricted objects, in: Proceedings of the 20th International Symposium on Distributed Computing, 2006, pp. 209–223.
[2] H. Attiya, J. Welch, Distributed Computing: Fundamentals, Simulations, and Advanced Topics, 2nd edition, John Wiley & Sons, 2004.
[3] M.J. Fischer, N.A. Lynch, M.S. Paterson, Impossibility of distributed consensus with one faulty process, J. ACM 32 (2) (1985) 374–382.
[4] M. Herlihy, Wait-free synchronization, ACM Trans. Prog. Lang. Syst. 13 (1) (1991) 124–149.
[5] M. Herlihy, N. Shavit, The Art of Multiprocessor Programming, Morgan Kaufmann, 2008.
[6] P. Jayanti, S. Toueg, Some results on the impossibility, universality, and decidability of consensus, in: Proceedings of the 6th International Workshop on Distributed Algorithms, 1992, pp. 69–84.
[7] S.A. Plotkin, Sticky bits and universality of consensus, in: Proceedings of the 8th ACM Symposium on Principles of Distributed Computing, 1989, pp. 159–175.