Eventual Wait-Free Synchronization

Tong che *

December 29, 2015

Abstract

Eventually linearizable objects are novel shared memory programming constructs introduced as an analogy to eventual consistency in message-passing systems. However, their behaviors in shared memory systems are so mysterious that very little general theoretical properties of them is known.

In this paper, we lay the theoretical foundation of the study of eventually linearizable objects. We prove that the n-process eventually linearizable fetch-and-cons (n-FAC) object is universal and can be used to classify the eventually linearizable objects. In particular, we define the concept of eventual consensus number of an abstract data type and prove that the eventual consensus number can be used as a good characterization of the synchronization power of eventual objects. Thus we got a complete hierarchy of eventually linearizable objects, as a perfect analogy of the consensus hierarchy. In this way, the synchronization power of eventual linearizability become much more well understood.

1 INTRODUCTION

Shared memory objects are the central programming constructs in concurrent computing. In essence, they provide high level abstractions to low level resources which make concurrent programs much easier to write, specify, and verify. Usually, these objects should be invoked by the processes independently, nevertheless their outcome can be explained from a sequential viewpoint. This property, called linearizability [4], is the most widely used criterion of correctness for shared memory objects. Linearizability is a safety property, which means any shared memory object that does not make any progress is trivially linearizable. A widely used liveness property is wait-freedom [3]. Wait-freedom guarantees all processes can make progress in bounded number of steps independently of other processes. For example, if a test-and-set linearizable wait-free object is accessed by a number of processes, there is a number $N > 0$, such that all operations invoked by the processes terminate in $N$ steps, and there is only one operation successfully set the state of object to one.

*Affiliation: IHES, France; Note: An update version of this paper will be submitted to PODC 2016
Shared memory objects which satisfy both linearizability and wait-free are important and thus attract a lot of attention. A fundamental result in shared memory computing [3] is that wait-free linearizable objects can be classified in terms of their synchronization power, by the notion of consensus number. Generally, a wait-free linearizable object \( O \) is of consensus number \( n \), if at most \( n \) processes can wait-free implement the consensus object with \( O \) and registers.

On the other hand, designing shared objects ensuring both linearizability and wait-freedom remains a hard task. So scientists try to weaken wait-freedom or linearizability to reduce this difficulty. By weakening wait-freedom, we can get objects with weaker liveness properties, such as lock-freedom, obstruction-freedom or deadlock-freedom. These notions are all useful in some particular contention level and system environments. Another research direction is to weaken the safety property, i.e., linearizability. In message-passing systems setting, eventual consistency [7] is such a weaker version of linearizability.

Several recent work in distributed computing tried to give a formal definition to eventual consistency. For example, in [5] [1], the abstract properties of eventual consistency models are defined. In [6], the definition of eventual linearizability is first introduced, and the problem of how to implement eventually linearizable objects with weakest failure detectors are studied. In [2], the strength and weakness of eventually linearizable objects are addressed by some notable examples, such as fetch-and-add or consensus objects. However, little general theory about the synchronization power of these objects are provided. The paradox of eventually linearizable shared objects are still a major mystery to the distributed community.

Roughly speaking, eventually linearizable objects can behave rather unpredictable at beginning, but should stabilize eventually after some steps after the failure. It is showed that eventual linearizability is strictly weaker than the usual notion of linearizability. However, some eventually linearizable objects such as consensus are trivial to implement, while some others are very hard to implement, at least as hard as their linearizable counterparts. However, the reason behind this phenomenon is to a large extent unknown. Especially, the synchronization power of wait-free eventually linearizable objects is an important open problem. It is asked [2] that whether there is a hierarchy of wait-free eventually linearizable objects of different synchronization power, as the case for linearizable objects. This paper answers these open problems affirmatively.

We prove that for each eventually linearizable objects, there is a quantity defined by \( n \text{\ fetch\&\ consensus} \) objects which serves as an analogy for consensus number. We call this quantity \textit{eventual consensus number}. In the eventual linearizability setting, the \textit{fetch\&\ consensus} objects are universal, in the sense that in a shared memory system with \( n \)-processes, wait-free eventually linearizable \textit{fetch\&\ consensus} objects can wait-free implement any eventually linearizable objects. Thus we get a novel classification of eventual linearizable objects by their synchronization power.

Our work is in a sense orthogonal to all the previous works on eventual linearizability. We lay the foundation of the study of eventually linearizable shared objects, by explicitly characterizing the synchronization power of different eventually linearizable objects, using eventual consensus number. We also
compute the eventual consensus number for a number of widely used types of shared objects.

2 MODELS AND DEFINITIONS

A shared memory system consists of a collection of processes, communicate with each other by accessing a number of shared objects of different abstract data types. The shared objects are accessed by invoking associated invocations. Abstract data types are defined by sequential specifications, formally, they are defined by a labelled transition system \( T \) consisting of:

1. A (possibly infinite) set of states \( S \).
2. A set of possible initial state \( S_0 \).
3. A set \( \mathcal{I} \) of possible object invocations.
4. A set \( \mathcal{V} \) of possible return values.
5. A labelled transition relation \( \delta \subseteq S \times \mathcal{I} \times \mathcal{V} \times S \).

In this paper, we consider mainly deterministic data types, namely, \( \delta \) is also a map \( \delta : S \times \mathcal{I} \rightarrow \mathcal{V} \times S \).

A shared memory system is composed of processes \( P = \{p_1, p_2, \ldots, p_n\} \) and a set of objects \( \{o_1, o_2, \ldots, o_m\} \). A event in the system is a tuple \((i, o, x)\), where \( i \) is the process identifier, \( o \) is the object accessed by the process, and \( x \) be the corresponding invocation/response. A history \( H \) is a sequence of events, and for each object \( o \) or process \( p \), we use \( H|o \) and \( H|p \) to denote the subsequence of history \( H \) consisting of events performed at object \( o \) and events performed by \( p \).

Definition 1 A history is called sequential, if it starts with a invocation event, and each invocation event \((p, o, i)\), expect the last event, is immediately followed by a corresponding response event \((p, o, r)\), no matter what \( i, r \) is.

A history \( H \) is called well-formed, if for each process \( p \), \( H|p \) is sequential. An operation is consists of two events: the invocation and the corresponding response.

A sequential history \( H \) is called legal, if for each object \( o \), \( H|o \) satisfy the sequential specification of \( o \).

An implementation of an object of type \( T = (S, S_0, \mathcal{I}, \mathcal{V}, \delta) \) is defined as a program each processes follows to perform operations in \( \mathcal{I} \). The program is a Turing machine that can do any local computations, can communicate with each other by invoking the underlying shared memory objects, and will finally return a result in \( \mathcal{V} \). In a concurrent execution of the implementation, many processes may repeatedly execute the program to perform different operations, and the steps of these operations may interleave each other in an arbitrary way.
**Definition 2** An implementation of a shared memory object is called wait-free if for every operation, it can terminate in finite number of steps, regardless of other processes. An implementation is called lock-free if some processes are invoking operations and taking steps, there will be at least one operation completes in finite number of steps.

Now we define the concept of eventual linearizability, following the same line of [2]. Eventual linearizability is roughly defined as weak consistency combined with $t$-linearizability for some $t$.

Weak consistency is defined as “solo-consistency”, which is a very weak safety property for distributed systems.

**Definition 3** A well-formed history $H$ is weakly consistent if for each operation $o_1$ (performed by $p_1$) that has a response in $H$ there is a legal sequential history $S$, such that

1. $S$ only contains operations invoked in $H$ before $o_1$ returns.
2. $S$ contains all operations that precede $o_1$ and are performed by $p_1$.
3. ends with the same response for $o_1$ in $H$.

We call $S$ the explanation history of operation $o_1$.

The notion of $t$-linearizability is a formal description of the idea that the system stabilizes after the first $t$ events.

**Definition 4** Let $t \geq 0$ be a natural number, $T$ be an abstract data type, and let $H$ be a well-formed history. Let $H_t$ be the suffix of $H$ obtained by removing first $t$ events. Then a legal sequential history $H_S$ with respect to $T$ is called a $t$-linearization of $H$ if

1. every operation invoked in $H_S$ is also invoked in $H$.
2. every operation completed in $H$ is also completed in $H_S$.
3. if the response of $o_1$ precedes the invocation of $o_2$, both of these events are in $H_t$ and $o_2$ is in $H_S$, then $o_1$ precedes $o_2$ in $H_S$.
4. If operation $o_1$ has a response in $H_t$, it also has a response in $H_S$, and both responses are the same.

**Definition 5** A well-formed history is $t$-linearizable if it has a $t$-linearization. A history $H$ is eventually linearizable if it is weakly consistent, and there is a $t \geq 0$, such that $H$ is $t$-linearizable. An implementation of an object with of type $T$ is eventually linearizable if every history it produces is eventually linearizable with respect to type $T$.  

4
3 EVENTUAL CONSENSUS NUMBERS

We define the concept of eventual consensus number, and then prove some basic properties of the wait-free eventual linearizability hierarchy. Unlike consensus number, which definition is associated with consensus objects, the eventual consensus number is associated with eventual fetch-and-cons objects.

3.1 Fetch-And-Cons Objects.

Informally, the fetch-and-cons (FAC) data type provides a single operation, fac(v). This operation adds v to the head of the list and return the entire list stored before the operation. Formally, the state of a FAC object consists of a list of values L, and the specification can be written as:

Precondition : \([L = L_0]\)
ret = fac(v)
Postcondition : \([\text{ret} = L_0 \&\& L = L_0 : v]\)

It is well known that the FAC object in classical sense has consensus number \(\infty\). We denote \(n - \text{FAC}\) as \(n\)-process fetch-and-cons object.

We study the circumstances under which it is possible to implement eventually linearizable FAC objects, with linearizable read/write registers. The reason why we use linearizable registers instead of eventually linearizable ones, is that it simplifies our arguments and proofs. But there is no essential difference. Everything can be extended to the case of eventually linearizable r/w registers.

For convinience, we sometimes call “eventually linearizable objects of type \(T_1\) with linearizable registers can wait-free implement eventual linearizable objects of type \(T_2\)” as “\(T_1\) wait-free eventually implements \(T_2\)”.

Definition 6 The eventual consensus number \(e^r_m(T)\) of data type \(T\) is the largest number \(n\) for eventually linearizable objects of type \(T\) and linearizable registers to implement \(n\)-process eventually linearizable FAC objects.

First, we prove the following lemma, about the basic property of \(e^r_m(T)\).

Lemma 1 Consider two abstract data types \(T_1\) and \(T_2\). If \(e^r_m(T_1) < e^r_m(T_2)\), then there exists no wait-free eventual linearizable implementation of \(T_2\) with eventual linearizable objects with type \(T_1\) in a system of more than \(e^r_m(T_1)\) processes.

Proof By contradiction, we assume \(T_1\) wait-free eventually implements \(T_2\) in a system of \(n\) processes, \(n > e^r_m(T_1)\). However, \(T_2\) wait-free eventually implements eventually linearizable \(n\)-process FAC. So if we substitute \(T_2\) with its implementation by \(T_1\), we get a eventually linearizable \(n\)-FAC implementation by \(T_1\). This is a contradiction to the definition of \(e^r_m(T)\). □
### 3.2 Stable Nodes and Eventual Implementations.

In order to prove some interesting properties of eventually linearizable objects, we introduce the concept of semi-stable and stable nodes, similar to [2]. We start with the definition of execution trees and stable nodes.

**Definition 7** Let $O_n$ be a wait-free eventually linearizable object shared by $n$ processes. The execution tree $T(O_n)$ is a tree induced by all the possible executions of $O_n$, with each edge representing an invocation or response event. Assume a node $C$ is on the execution tree, and $l(C)$ is the path from the tree root down to $C$. We denote $|l(C)|$ the length of the path. $C$ is called a stable node if every possible execution of $O_n$ with prefix $l(C)$ is $|l(C)|$-linearizable.

The following lemma can be viewed as a generalization of Lemma 17 in [2]. It shows that under the condition of eventual linearizability, the $t$-linearizability of FAC object becomes a safety property.

**Lemma 2** Suppose we have an eventually linearizable implementation of a FAC object. Let $t > 0$ be a natural number, and let $\alpha$ be an infinite history of this implementation. If every finite prefix of $\alpha$ is $t$-linearizable, then $\alpha$ is $t$-linearizable.

**Proof** Without loss of generality, we assume that there is $t'$, such that $\alpha$ has $t'$-linearization $S'$. We can further assume $t' > t$, because if $t' \leq t$, there is nothing to prove.

We classify the operations in $\alpha$ into four subsets:

- $S_1 = \{\text{operations with response among the first } t \text{ events of } \alpha.\}$
- $S_2 = \{\text{operations with response among events } t + 1, t + 2, \ldots, t' \text{ of } \alpha.\}$
- $S_3 = \{\text{operations with response after } t' \text{ events of } \alpha.\}$
- $S_4 = \{\text{operations that do not complete in } \alpha.\}$

We first consider any two operations $o_1$ and $o_2$ in $S_2 \cup S_3$. Assume their corresponding response values are $l_1$, $l_2$. Consider the $t$-linearization of a prefix of $\alpha$ containing both $o_1$ and $o_2$. According to the definition of $t$-linearizability, in this legal sequential history, $o_1$ and $o_2$ are still completed with responses $l_1$ and $l_2$. So in the two lists $\{l_1, l_2\}$, one must be a strict prefix of the other. Without loss of generality, we assume $l_1$ is the strict prefix of $l_2$. We can also write $l_1 < l_2$, and $o_1 < o_2$. It follows that the operations in $S_2 \cup S_3$ are totally ordered, according to their response value.

Our main task is to construct the $t$-linearization $S$ of $\alpha$. If an operation $o \in S_2 \cup S_3$ has a response $l$, we assign the operation to the $|l|$-th slot in $S$. Then the assigned operations form a chain, in which former operations in the chain are strictly prefixes of the later operations. There is the set $E$ of empty slots that have not yet been assigned to operations, before the last slot assigned to operations in $S_2 \cup S_3$. If we assign all the operations in $S_1$ and some operations
in $S_4$ to $E$ with legal responses, we can get a $t$-linearization of $\alpha$. Note the “legal” responses for the empty slots are already fixed by the prefix relation.

Write

$$E = \{ l | l \text{ is a prefix for some response } l' \text{ in } S_2 \cup S_3, \text{ but } l \text{ is not a response in } S_2 \cup S_3 \}$$

This set has a one to one correspondence with the set of empty slots $E$. We to prove that $|S_1| \leq |E| \leq |S_1| + |S_4|$. We consider the $t'$-linearization $S'$, and it is obvious that any operation $o$ in $S_3$ are assigned to $|l|$-th slot in $S'$ if $l$ is the response of $o$. Let

$$E' = \{ l | l \text{ is a prefix for some response } l' \text{ in } S_3, \text{ but } l \text{ is not a response in } S_3 \}$$

Then $E'$ is the set of slots that has not been assigned, and $S'$ fills all the slots of $E'$ with operations in $S_1$, $S_2$, and some subset $S'_4 \subseteq S_4$. So we have $|S_1| + |S_2| \leq |E'| \leq |S_1| + |S_2| + |S_4|$. Each operation has been assigned to the same slot by $S$ and $S'$. So $|E| = |E'|-|S_2|$. So we proved $|S_1| \leq |E| \leq |S_1| + |S_4|$.

We now show how to construct linearization $S$. Since both $S_1$ and $S_4$ are finite sets, we consider the last empty slot $l'_m$ in $E$. Since we assume the last event in $S$ is in $S_2 \cup S_3$, $l'_m$ is followed by a operation with response $l_m$ in $S_2 \cup S_3$. Consider the operation $o_m$ with response $l_m$. Consider the $t$-linearization of the finite history from beginning to the response of $o_m$, which we denote as $S_m$. $S_m$ coincides with $S$ on all slots assigned for operations in $S_2 \cup S_3$, but fills all the empty slots with operations in $S_1$ and some operations in $S_4$. It is obvious to see by simply extending $S_m$ with all the operations in $S$ after $o_m$, we can get a $t$-linearization $S$ of $\alpha$. 

\[\square\]

### 3.3 Hardness of Implementation of Eventual n-FAC Objects.

Now we can prove a proposition about the hardness of implementation of wait-free eventually linearizable $n$-FAC objects from linearizable objects. This proposition basically means that implementing wait-free eventually linearizable $n$-FAC objects is as hard as implementing their linearizable counterparts.

**Proposition 1** If there is a $n$-process eventually linearizable, wait-free implementation of a fetch-and-cons object from a set $O$ of linearizable objects, then there is a $n$-process linearizable wait-free implementation of a fetch-and-cons object from $O$.

**Proof** Let $A$ be an eventually linearizable wait-free implementation of $n$-FAC object from $O$. Consider the execution tree $T(A)$ of $A$, each of the $n$ processes repeatedly perform fac() operations forever. We first prove the following claim:

Claim: There is at least one stable node on the tree $T(A)$.

Proof of Claim: We prove the claim by contradiction. If there is no stable node, we can inductively construct a sequence of finite paths $p_1, p_2, \cdots$ that
having the following properties, where \( l_i = |p_0p_1 \cdots p_i| \) is the length of the concatenated path.

- For \( i \geq 0 \), \( p_0p_1 \cdots p_i \) is a connected path starting from the root of the tree.
- For \( i \geq 1 \), \( p_i \) is not empty.
- For \( i \geq 1 \), \( p_0p_1 \cdots p_i \) is not \( l_{i-1} \)-linearizable.

Let \( p_0 \) be the empty execution. For the inductive step, let \( i \geq 1 \), we are given \( p_1, p_2, \ldots, p_{i-1} \) satisfying the above properties. We now construct \( p_i \) as follows.

Let \( C \) be the ending node of the path \( p_1 p_2 \cdots p_{i-1} \). \( C \) is not stable, so there is some execution \( \alpha \) with prefix \( p_1 p_2 \cdots p_{i-1} \) that is not \( l_{i-1} \)-linearizable. So due to Lemma 9, some finite prefix \( \alpha' \) is not \( l_{i-1} \)-linearizable. Let \( p_i \) be the path such that \( \alpha' = p_1 p_2 \cdots p_i \).

The infinite execution \( \pi = p_0p_1 \cdots \) much be \( t \)-linearizable for some \( t \).

Choose \( i \) such that \( l_{i-1} > t \), so \( \pi \) is \( l_{i-1} \)-linearizable. This contracts the above fact that its finite prefix \( \alpha' \) is not \( l_{i-1} \)-linearizable. So the claim is true.

Let \( C \) be a stable node on \( T(A) \). We denote \( \alpha_C \) the finite execution history from the root to node \( C \). Consider the configuration \( C_{idle} \) reached from \( C \) by letting each process run solo to complete its current operation. Then we let one process \( p \) run the operations fac() repeatedly. It is easy to conclude from the definition of eventual linearizability that eventually some operation \( o_0 \) that \( p \) performs must return a list of values that is equal to the set of all values inserted to the list before \( o_0 \). Let \( C_0 \) be the configuration at the end of \( o_0 \). Let \( l_0 \) be the response of \( o_0 \).

Then we can construct a linearizable implementation \( A' \) from \( O \). First, initialize objects in \( A \) to \( C_0 \), and then, if a \( A'.fac(v) \) operation is invoked, the algorithm \( A' \) just execute algorithm \( A \), and when a response \( l = A.fact(v) \) is ready, algorithm \( A' \) just respond with \( l \setminus l_0 \).

It is trivial to verify \( A' \) is a linearizable implementation.

\[ \square \]

### 3.4 Relationship With Classical Consensus Number.

For any abstract data type \( T \), we have two numbers. The classical consensus number \( c(T) \) characterizes the synchronization power of wait-free linearizable shared objects of type \( T \), while the eventual consensus number \( e^r_m(T) \) characterizes the synchronization power of their eventually linearizable counterparts. We have the following theorem, which give us strong information of the number \( e^r_m(T) \), and plays an essential rule in the computation of eventual consensus numbers.

**Theorem 1** The eventual consensus number induces a hierarchy of abstract data types. For any abstract data type \( T \), we have \( e^r_m(T) \leq c(T) \).
Proof It suffices to prove that a list of eventually linearizable objects of type $T$ together with linearizable registers cannot implement eventually linearizable $(c(T) + 1)$-FAC object. We prove this by contradiction. Assume a list of linearizable objects of type $T$ together with registers can wait-free implement eventually linearizable $(c(T) + 1)$-FAC object. From Proposition 10, a list of linearizable objects of type $T$ can wait-free implement linearizable $(c(T) + 1)$-FAC object, which is universal. So a list of linearizable objects of type $T$ can wait-free implement linearizable $(c(T) + 1)$-consensus. This is a contradiction. $\blacksquare$

**Corollary 1** The eventual consensus number for read/write registers is 1.

### 3.5 Universal Construction.

The classical consensus objects are universal, in the sense that using $n$-process consensus, we can implement any Turing computable abstract data types shared by $n$ process system, by the well-known universal construction.

We show in this section that we can construct the universal construction in the eventual linearizability sense, by using $n$-FAC objects instead of consensus. In particular, we can prove the following theorem:

**Theorem 2** For any abstract data type $T$, any eventually linearizable object of type $T$ shared by $n$ processes can be wait-free implemented by a single copy of eventually linearizable $n$-FAC.

**Proof** We consider the following universal construction for any object of abstract data type $T$:

**Algorithm : Universal Construction For FAC**

Require: $O_{\text{local}}$, a local copy of object of Type $T$.
Require: Operation, the operation ID that is invoked.
Require: Parameter, the parameters of the operation that is invoked.
Invoke(Operation, Parameter):

- $l_{\text{local}} := \text{fac(Operation, Parameter)}$;
- $O_{\text{local}}.\text{Initialize}()$;
- for each $(op, m)$ in $l_{\text{local}}$:
  - $O_{\text{local}}.op(m)$;
- end;
- $v := O_{\text{local}}.\text{Operation(Paramter)}$;
- return $v$;
- end;

It is obvious that the above algorithm wait-free implements any eventually linearizable objects shared by $n$ processes.

The proof is almost trivial. Since the FAC object is eventually linearizable, for any history of the above algorithm implementing an object $O$, the
explanation history of any operation of \( \mathcal{O} \) is one to one corresponding to the explanation history of the FAC response. An \( t \)-linearization of the FAC history also easily lead to the \( t \)-linearizability of history of \( \mathcal{O} \). So both weak consistency and \( t \)-linearizability follow. □

4 EVENTUALLY LINEARIZABLE SHARED OBJECTS

In this section, we apply the theory to the study of a list of eventually linearizable shared objects of some common abstract data types.

4.1 One-Shot Shared Objects

There has been an intuition that “one-shot” shared memory objects are trivial in eventually linearizable settings [2]. For example, it is proved that eventually linearizable consensus objects can be implemented from eventually linearizable registers. However, here we provide the first general explanations of why they are weak, using our concept of eventual consensus number.

4.1.1 Consensus Object

We have the following theorem for the eventual consensus number of the consensus data type.

**Theorem 3** Let \( T_{\text{cons}} \) be the consensus data type, we have \( e^*_m(T_{\text{cons}}) = 1 \).

**Proof** We first prove that eventually linearizable consensus can be implemented from registers. We have the following implementation:

**Algorithm : Implementation of Consensus**

- **Require:** Process ID \( id \), one register \( \text{Proposal}[i] \) for each process \( i \), initialized with NULL.
- **Require:** proposal value \( v \).
- **Propose**(\( v \))
  - if \( \text{Proposal}[i] = \text{NULL} \)
    - \( \text{Proposal}[i] = v \);
  - end;
  - for \( k = 1 \cdots n \)
    - if \( \text{Proposal}[k] \neq \text{NULL} \)
      - return \( \text{Proposal}[k] \);
  - end; end; end;

The implementation is obviously wait-free. To justify its weak consistency, we only need to say every process will return some value and every value it
returns was proposed by some processes. Also, in any history \( H \) of the imple-
m entation above, there must be a natural number \( t \), such that the contents of
Proposal\([i]\) do not change after \( t \) events, for all \( i \). It is then trivial to see that
the history \( H \) is \( t \)-linearizable.

Then we prove by contradiction, assume \( e_m^r(T_{cons}) > 1 \). Then eventually
linearizable consensus object can implement 2-FAC object. Then linearizable
registers can implement linearizable 2-FAC, due to Proposition 10. This is a
contradiction, so \( e_m^r(T_{cons}) = 1 \).

\[ \square \]

**4.1.2 Test-and-Set Object**

The test-and-set object is another example of one-shot object. In particular, we
have:

**Theorem 4** The test-and-set data type has eventual consensus number 1.

**Proof** According to the proof of Theorem 14, it suffices to give an implemen-
tation of a test-and-set object using registers. We give one as fo llows.

**Algorithm : Implementation of TAS**

**Require:** one register \( r \), initialized with 0.

**TAS()**

\[
\begin{align*}
\text{if } r = 0 \\
\quad r := 1; \text{ return True}; \\
\text{else return False;}
\end{align*}
\]

The above implementation is very simple, it just uses a register instead of
the test-and-set object. It is also easy to prove this is a eventua lly linearizable
TAS object. So we are done.

\[ \square \]

**4.2 Long-lived Shared Objects**

In contrast to one-shot objects, long lived objects can modify their internal
states forever. It is natural to ask whether some common long lived objects
have positive eventual consensus numbers. In the current version of the paper,
we focus on one such data type — the fetch-and-add data type. We have the
following theorem:

**Theorem 5** The fetch-and-add data type has eventual consensus number 2.

**Proof** Let \( T \) be the fetch-and-add data type. From theorem 11, we know that
\( e_m^r(T) \leq 2 \). It then suffice to prove that \( e_m^r(T) \geq 2 \). In particular, we give an
implementation of eventually linearizable 2-FAC using fetch-and-add objects in
a two process system.

**Algorithm Implementation of 2-FAC**

11
Require: Process ID id, an array of registers for each process sequence.id[], id=1 or 0.

Require: an array of registers for each process index.id[]

Require: Fetch-and-add object $F$, initialized to 0.

Require: one counter.id for each process id, initialized to 0.

Fetch:\text{And\_Cons}(value)
\begin{align*}
&\text{sequence.id}[\text{counter}] := \text{value}; \\
&\text{ind} := F.\text{fetch\_and\_add}(); \\
&\text{index.id}[\text{counter.id}] := \text{ind}; \\
&\text{list} := \text{Reorder\_List}(\text{sequence}, \text{index}, \text{ind}); \\
&\text{counter.id} := \text{counter.id} + 1; \\
&\text{return list};
\end{align*}

\text{Reorder\_List}(\text{sequence}, \text{index}, \text{ind}) : \\
\text{list} = \text{merge}(\text{sequence}, \text{index}); \\
\text{/* first fill the list with elements in sequence.0[i] in slot index.0[i], then fill the rest slots using sequence[i], in sequential order*/} \\
\text{return first}_\text{ind}_\text{elements(list, ind)}; // cut the list with first ind elements

It is easy to derive the eventual linearizability of above algorithm. \qed

References

[1] Sebastian Burckhardt, Alexey Gotsman and Hongseok Yang. Understanding eventual consistency. Technical Report MSR-TR-2013-39, March 2013.

[2] Rachid Guerraoui and Eric Ruppert. A paradox of eventual linearizability in shared memory. In Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing, PODC ’14, pages 40–49. New York, NY, USA, 2014. ACM.

[3] Maurice Herlihy. Wait-free synchronization. ACM Trans. Program. Lang. Syst., 13(1):124–149, jan 1991.

[4] Maurice P Herlihy and Jeannette M Wing. Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems (TOPLAS), 12(3):463–492, 1990.

[5] Yasushi Saito and Marc Shapiro. Optimistic replication. ACM Comput. Surv., 37(1):42–81, mar 2005.
[6] Marco Serafini, Dan Dobre, Matthias Majuntke, Péter Bokor and Neeraj Suri. Eventually linearizable shared objects. In Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, PODC ’10, pages 95–104. New York, NY, USA, 2010. ACM.

[7] Werner Vogels. Eventually consistent. Commun. ACM, 52(1):40–44, jan 2009.