The $F$-snapshot Problem.

Gal Amram
Department of Computer Science,
Ben-Gurion University, Beer-Sheva, Israel

Abstract

Aguilera, Gafni and Lamport introduced the signaling problem in [5]. In this problem, two processes numbered 0 and 1 can call two procedures: update and Fscan. A parameter of the problem is a two-variable function $F(x_0, x_1)$. Each process $p_i$ can assign values to variable $x_i$ by calling update($v$) with some data value $v$, and compute the value: $F(x_0, x_1)$ by executing an Fscan procedure. The problem is interesting when the domain of $F$ is infinite and the range of $F$ is finite. In this case, some “access restrictions” are imposed that limit the size of the registers that the Fscan procedure can access.

Aguilera et al. provided a non-blocking solution and asked whether a wait-free solution exists. A positive answer can be found in [7]. The natural generalization of the two-process signaling problem to an arbitrary number of processes turns out to yield an interesting generalization of the fundamental snapshot problem, which we call the $F$-snapshot problem. In this problem $n$ processes can write values to an $n$-segment array (each process to its own segment), and can read and obtain the value of an $n$-variable function $F$ on the array of segments. In case that the range of $F$ is finite, it is required that only bounded registers are accessed when the processes apply the function $F$ to the array, although the data values written to the segments may be taken from an infinite set. We provide here an affirmative answer to the question of Aguilera et al. for an arbitrary number of processes. Our solution employs only single-writer atomic registers, and its time complexity is $O(n \log n)$, which is also the time complexity of the fastest snapshot algorithm that uses only single-writer registers.

1 Introduction

In this paper we introduce a solution to the $F$-snapshot problem, which is a generalization of the well-studied snapshot problem (introduced independently by Afek et al. [2] [3], by Anderson [8] and by Aspnes and Herlihy [9]). A snapshot object involves $n$ asynchronous processes that share an array of $n$ segments. Each process $p_i$ can write values to the $i$-th segment by invoking an update procedure with a value taken from some range of values: $Vals$, and can scan the entire array by invoking an instantaneous scan procedure. For any function $F : Vals^n \rightarrow D$ (where $D$ is any set and $Vals^n$ is the set of $n$-tuples of members of $Vals$) the $F$-snapshot variant differs from the snapshot problem in that the Fscan operation has to return the value $F(v_0, \ldots, v_{n-1})$ of the instantaneous segment values $v_0, \ldots, v_{n-1}$. That is in comparison to the standard scan operation, which returns the vector of values that the segments store at an instantaneous moment.

The $F$-snapshot problem is interesting only if we impose an additional requirement, without which it can be trivially implemented by applying the function $F$ (assumed to be computable) to the values returned by the standard scan operation. This additional requirement, for the case $n = 2$, was suggested by Aguilera, Gafni and Lamport [5] (see also [4]) in what they called there the signaling problem. Thus, our $F$-snapshot problem is a generalization of both the standard snapshot problem and the signaling problem (generalizing this problem from the $n = 2$ case to the general case of arbitrary $n$).

In the signaling problem the set $Vals$ can be assumed to be infinite, and the set $D$ (the range of $F$) is finite (and small). The requirement is that an Fscan operation uses only bounded registers. That is, registers that can store only finitely many different values (the update operations may access unbounded registers). The signaling problem was formulated just for two processes in [5], and a wait-free solution for this problem was left there as an open problem. Thus, solving the general $F$-snapshot sets quite a challenge. A wait-free
solution to the signaling problem is given in [7], and here we present a (wait free) solution to the general $F$-snapshot problem.

As the domain of $F$ may be infinite, an update operation cannot access only bounded registers. Furthermore, the $F$-snapshot problem generalizes the signaling problem which provides a solution to the mailbox problem. Abraham and Amram [1] showed that even the mailbox problem cannot be solved while only bounded registers are employed.

In [5], the signaling problem is justified for efficiency reasons. We consider a case in which the processes write values to their segments taken from an infinite range, but they are interested in some restricted data regarding these values (for example, which process invoked the largest value, how many different values there are etc.). An $F$-snapshot implementation may be more efficient in these cases than a snapshot implementation, since it is not necessary to scan the entire array for extracting the required information, and it suffices to read only bounded registers. Efficiency is mostly guaranteed when the Fscan operations are likely to be invoked much more frequently than the update procedures.

Now we describe the $F$-snapshot problem formally. Let $P = \{p_0, \ldots, p_{n-1}\}$ be a set of $n$-asynchronous processes that communicate through shared registers and let

$$F : \text{Vals}^n \rightarrow D$$

be an $n$-variables computable function from a (possibly infinite) domain Vals, into $D = \text{Rng}(F)$. The problem is to implement two procedures:

1. **update**($v$) - invoked with an element $v \in \text{Vals}$. This procedure writes $v$ to the $i$-th segment of an $n$-array $A$, when invoked by $p_i$.

2. **Fscan** - returns a value $d \in D$. This procedure returns $F(A[0], \ldots, A[n - 1])$, in contrast to a scan procedure which returns the entire array: $(A[0], \ldots, A[n - 1])$.

The implementation needs to satisfy the following requirements:

1. All procedures are wait free. That is, each procedure eventually returns, if the executing process keep taking steps.

2. If $D$ is finite, then only bounded registers are accessed during Fscan operations.

The $F$-snapshot problem can be studied under various communication restrictions. As an example, the $f$-array implementation by Jayanti [22] solves the $F$-snapshot problem as well, when the LL/SC primitive is employed. However, the LL/SC operation cannot be implemented from read/write operations [15]. Here we assume the simplest means of communication:

3. Only single-writer multi-reader atomic registers are applied.

For correctness of $F$-snapshot implementations, we adapt the well known Linearizability condition, formulated by Herlihy and Wing [17]. Roughly speaking, an $F$-snapshot algorithm is correct if for any of its executions the following hold: Each procedure execution can be identified with a unique moment during its actual execution (named the linearization point), such that the resulting sequential execution belongs to a set of correct sequential executions: the sequential specification of the object. The sequential specification of the $F$-snapshot object includes all executions of the following atomic implementation, presented by a code for process $p_i$. The code uses an array $A[0..n-1]$.

\begin{align*}
\text{update}(v) & \quad \quad \quad \text{Fscan()}
\begin{align*}
1. & \quad A[i] := v \quad \quad \quad 1. & \quad \text{return } F(A[0], \ldots, A[n - 1])
\end{align*}
\end{align*}
One of the basic methods for proving linearizability is to identify each procedure execution with the execution of one of its actions, and to prove that these linearization points satisfy the requirements. However, in some cases the linearization points are not fixed, and may even be identified with actions executed concurrently by other processes (the queue implementation in [17] forms an example of such an algorithm). Hence, this approach is not complete. Therefore, we present the linearizability condition in an equivalent way to the one described above, a way that fits better the correctness proof we provide here for our $F$-snapshot algorithm.

In an execution of an $F$-snapshot algorithm, the procedure executions are partially ordered by the precedence relation $\prec$. That is, if $A$ and $B$ are procedure executions, $A \prec B$ means that $A$ ends before $B$ begins. An execution is linearizable if the relation $\prec$ can be extended to a linear ordering $\preceq$ that satisfies the sequential specification, presented in Figure 1. An $F$-snapshot algorithm is correct if all its executions are linearizable.

1. All procedure executions are partitioned into update and $F$scan operations. An update operation is invoked with a value $v \in Vals$ and an $F$scan returns a value $d \in D$.
2. Each procedure execution belongs to a unique process $p_i$, $i < n$.
3. Let $S$ be an $F$scan operation and for each $i < n$ assume that $U_i$ is the last $p_i$-update event that precedes $S$ in $\prec$. Assume that each $U_i$ is invoked with a value $v_i$. Then, $S$ returns $F(v_0, \ldots, v_{n-1})$.

Figure 1: $F$-snapshot sequential specification

In this paper, we present a solution to the $F$-snapshot problem. Each operation in our algorithm consists of $O(n \log n)$ actions addressed to the shared registers. Thus, the time complexity of our algorithm equals the snapshot implementation by Attiya and Rachman [11] which, as far as we know, the fastest published snapshot implementation with single-writer registers.

2 The $F$-snapshot Algorithm

First we explain the main ideas behind the algorithm. The reader may want to consider our explanations, while examining the code of the algorithm given in Figure 2 and its local procedures in Figure 3.

The crucial obstacle for solving the problem is that an $F$scan procedure cannot access unbounded registers, but it is required to apply the function $F$ on values that are stored in unbounded registers. Thus, the computation of $F$ needs to be done during an execution of an update operation. When process $p_i$ performs an update operation invoked with a data value $val$, it writes $val$ into a snapshot object $V$ (line 2), scans this snapshot object (line 3), applies the function $F$ on the view it obtained and stores the outcome in a local variable $ans$ (line 4). Then, before it returns it writes the outcome it obtained into a snapshot object named $Flags$ (line 16). A scanner scans the snapshot object $Flags$ and it needs to choose the most up-to-date value among the values suggested by the processes. We need to provide the $Flags$ object with an additional information, so a scanner could decide correctly which value to return. However, this additional information needs to be taken from a finite range due to the problem limitations.

As a first attempt, one may suggest to use bounded concurrent timestamps [19] to label update events (see [12], [13], [14]). The intuitive approach is to choose a timestamp after updating $V$ and before scanning $V$, or to use a $scate$ operation as in [11] and to choose a timestamp immediately after the $scate$ operation returns. The problem is that a scanner will return a value delaying on the order between the labeling operations and not on the order between the scan events addressed to the snapshot object $V$. Hence, this approach in its simplest form, will not succeed.
Our goal is to provide the Flags snapshot object with some additional bounded information, so that a scanner will be able to determine the right order between scans addressed to V that precede the writes into the snapshot object Flags. Several synchronization mechanisms are used for achieving this goal.

2.1 The Classify Mechanism

For determining the ordering between scan events addressed to V, we adapt common technique of counting update events [11],[10],[20]. When a process performs an update operation, it increases a counter (line 1) and writes this counter to V together with the data value with which the update operation was invoked (line 2). After a process scans V, it sums these counters to obtain a natural number that reflects how recent its view is (line 9). This approach resembles the snapshot algorithm presented by Israeli, Shaham and Shirazi [20]. They used this technique to implement a snapshot algorithm in which the time complexity of the scan procedure is O(n). In their construction, while executing a scan operation, the executing process returns the view of the process that presents the latest activity, reflected by the largest sum.

Since an Fscan operation cannot access unbounded registers, we cannot adopt the discussed approach as it was used in [20]. In our algorithm, reading these natural numbers is done while executing an update procedure. The process writes the sum it obtained into a snapshot object named ViewSum (lines 10,12), and scans ViewSum to compare its view with the views obtained by the other processes. Afterwards, it classifies all other processes into two categories: winners - the processes that posses a later view, reflected by a largest sum, and losers - processes with less up-to-date view. This is done by calling the local classify procedure, when processes id’s are used for breaking symmetry. Eventually, these sets of winners and losers will be stored at the segment Flags[i] (lines 15,16).

2.2 The coloring Mechanism

A scanner scans the Flags array and it tries to find the most up-to-date view while considering the fields Flags[i].winner and Flags[i].losers for i = 0,...,n – 1. For any pair of processes pi and pj, the scanner tries to understand which process’s view is more recent. The problem is that the processes may provide contradicting information. As an example, a scanner may find that (j ∈ Flags[i].winners (which means that pi thinks that pj’s view is more up-to-date then its view), but it is possible that also i ∈ Flags[j].winners. Namely, it is possible that both pi and pj think that the other process knows better.

The coloring mechanism ensures that the problem described above can occur only in some “typical” executions (with which our next mechanism deals). The update events by each process alternate between 3 possible colors: 0, 1 or 2 (line 1). Each process posses a three-fields variable, in correspondence to three colors, named myview. After a process sums the counters it sees (lines 3,9), it writes the sum it obtained into myview|color (line 10) and deletes data obtained in its second-previous update operation (line 11) to erase a confusing information. Then, it writes the value that myview stores into ViewSum (line 12). Now, when process pi scans ViewSum, in each segment ViewSum[j], it finds two integers. These are the sums that pj computed in its two previous update events. When pi executes its local classify procedure, it also writes the color it saw. For example, it writes (j,c) to winners for c ∈ {0, 1, 2}, if it read from ViewSum[j][c] a number greater than the sum it obtained in line 9 (when id’s are taken into account for breaking symmetry). Each process pi writes the values of its local sets winners and losers to Flags[i], together with the color of the update operation it is executing.

Coming back to our example, now each process also specifies the color of the update operation it saw. If a scanner finds that (j,c) ∈ Flags[i].winners it understands that pi saw in ViewSum[j][c] an integer larger than the number it obtained. However, if the scanner sees that Flags[j].color ≠ c, it just disregards pi’s information.

2.3 Adding Bounded Timestamps

The coloring mechanism does not prevent entirely the possibility that processes will provide contradicting information. Assume as an example that a scanner finds that Flags[i].color = c₁, Flags[j].color = c₂,
\((j, c_j) \in \text{Flags}[i].\text{winners}\) and \((i, c_i) \in \text{Flags}[j].\text{winners}\). Thus, both \(p_i\) and \(p_j\) claim that the other process is more up-to-date. When such a situation occurs, one of the processes provides reliable information. This is the process that scanned \(\text{ViewSum}\) later before updating \(\text{Flags}\).

When such a situation occurs, the processes use timestamps to inform which process is trustworthy. We use a simple timestamps system in which the timestamps are vertices of a nine-vertices directed graph \(G = (V_G, E_G)\). A detailed explanation can be found in chapter 2 of [16] or in [19]. The graph \(G\) consists of three cycles, each cycle includes three vertices. In addition, there is an edge from each vertex at the \(i\)-th cycle to each vertex at the \(i - 1 \pmod{3}\) cycle. Formally, \(V_G = \{(i, j) : i, j \in \{0, 1, 2\}\}\), and there is an edge from \(v = (i_1, j_1)\) to \(u = (i_2, j_2)\) if \(i_1 = i_2\) and \(j_1 = j_2 + 1 \pmod{3}\), or \(i_1 = i_2 + 1 \pmod{3}\). The vertices of \(G\) are named timestamps, and if \((v, u) \in E\) we say that \(v\) dominates \(u\) and we write \(u <_ts v\). Intuitively, \(v\) dominates \(u\) means that the timestamp \(v\) represents a later moment than the timestamp \(u\).

We can see that there no cycles of length two in \(G\). In addition, for any two timestamps \(v, u\), we can find a timestamp \(w\) that dominates both \(v\) and \(u\). We take a function \(\text{next} : V_G \times V_G \rightarrow V_G\) that satisfies this property. That is, for any timestamps \(v, u\): \(v <_ts \text{next}(v, u)\) and \(u <_ts \text{next}(v, u)\).

Any process \(p_i\) holds \(n\) pairs of timestamps. Each pair consists of a new timestamp and an old timestamp. These pairs are stored in a snapshot object named \(\text{VTS}\). When \(p_i\) executes an \(\text{update}\) operation it scans \(\text{VTS}\) (line 5). Then, against each process \(p_j\) it chooses a timestamp that dominates the pair of timestamps it read from \(\text{VTS}[j][i]\), using the function \(\text{next}\). \(p_i\) stores the timestamp it obtained as its new timestamp, keeps its former timestamp available as its old timestamp and updates \(\text{VTS}\) (consider lines 5-8 and the local procedure \(\text{newts}\)). Finally, \(p_i\) stores its \(n\)-vector of pairs of timestamps in \(\text{Flags}[i]\) while updating the \(\text{Flags}\) object (line 16).

Now, consider again the situation in which a scanner finds that two processes \(p_i\) and \(p_j\), provide contradicting information as described earlier. In this case, the scanner checks the timestamps that the processes present. The process that its new timestamp dominates the other process’s new timestamp is the reliable one. More precisely, the scanner considers the timestamps \(\text{Flags}[i].\text{vts}[j].\text{new}\) and \(\text{Flags}[j].\text{vts}[i].\text{new}\). The information provided by the process with the later timestamp is the right information. These timestamps are used only when processes provide contradicting information. In other cases the timestamps do not necessarily reflect the right ordering between the processes’ views.

### 2.4 The Code

Now we specify the code of the algorithm and we start by presenting the data structures and the type of the registers and variables. First, the algorithm use 4 snapshot objects.

1. \(V\) - each entry \(V[i]\) stores a pair: \((n, val) \in \mathbb{N} \times \text{Vals}\). \(val\) is the value with which the \(\text{update}\) procedure is invoked, and \(n \in \mathbb{N}\) counts the number of \(\text{update}\) operations invoked by \(p_i\). Note that as these values are taken from an infinite range, an \(\text{Fscan}\) operation cannot access this object. Initially each segment stores the value \((0, x_0)\) for some fixed \(x_0 \in \text{Vals}\).

2. \(\text{VTS}\) - each entry \(\text{VTS}[i]\) stores an \(n\)-array: \(\text{vts}[i][0..n - 1]\) of pairs of timestamps. Thus, each entry \(\text{vts}[i][j] = (v, u)\) where \(v, u\) are timestamps. The first field is denoted \(\text{vts}[i][j].\text{old}\), while the second field is \(\text{vts}[i][j].\text{new}\) i.e. \((v, u) = (\text{vts}[i][j].\text{old}, \text{vts}[i][j].\text{new})\). Initially, each field \(\text{VTS}[i][j]\) stores \((v_0, v_0)\) for some fixed \(v_0 \in V_G\).

3. \(\text{ViewSum}\) - each entry \(\text{ViewSum}[i]\) is a triple: \(\text{viewsum}[i][0..2]\) of natural numbers when each entry may also store \(\text{null}\). That is, \(\text{ViewSum}[i] \in (\mathbb{N} \cup \{\text{null}\}) \times (\mathbb{N} \cup \{\text{null}\}) \times (\mathbb{N} \cup \{\text{null}\})\). There are three fields in correspondence to three possible colors of the \(\text{update}\) operations. The initial value of each segment \(\text{ViewSum}[i]\) is \((0, \text{null}, \text{null})\).

4. \(\text{Flags}\) - this is the bounded object scanned during \(\text{Fscan}\) operations. Each entry \(\text{Flags}[i]\) stores an element of type \(\text{flag}\). The \(\text{flag}\) type consists of five fields:
   
   (a) \(\text{flag.color} \in \{0, 1, 2\}\). Initially this field is 0.
(b) flag.vts - an n-array of pairs of timestamps. Initially all pairs are \((v_0, v_0)\). Recall that \((v_0, v_0)\) is also the initial value of each \(VTS[i][j]\).

(c) flag.winners, flag.losers - sets of pairs of the form: \((i, c) \in \{0, \ldots, n - 1\} \times \{0, 1, 2\}\). At the \(i\)-th segment flag.winners and flag.losers are initialized to \(\{(j, 0) : i < j\}\) and \(\{(j, 0) : j < i\}\) respectively.

(d) flag.ans \in D. The initial value of this field is \(F(x_0, x_0, \ldots, x_0)\). Recall that \(x_0 \in Vals\) is the initial value of each entry \(V[i]\).

Each process use several local variables:

1. color \in \{0, 1, 2\}. Initially color = 0.
2. counter, viewsum \in \mathbb{N}. The initial value of these variables is 0.
3. val \in Vals.
4. ans \in D.
5. myview \in (\mathbb{N} \cup \{null\}) \times (\mathbb{N} \cup \{null\}) \times (\mathbb{N} \cup \{null\})\). Initially myview = (0, null, null).
6. winners, losers - sets that contain elements from the range \(\{0, \ldots, n - 1\} \times \{0, 1, 2\}\).
7. vts[i][0..n - 1] - an n-array of pairs of timestamps.
8. ts.old, ts.new - timestamps.
9. Other variables that are used for storing information while scanning the snapshot objects (lines 3, 5, 13). The type of each such a variable is in correspondence to the type of the objects that are scanned.

| update(val) |
|----------------------------------|
| 1. counter := counter + 1, color := counter \mod 3, |
| 2. V.update(counter, val) |
| 3. \((v_0, \ldots, v_{n-1}) = V.scan\) |
| 4. ans := F(v_0.val, \ldots, v_{n-1}.val) |
| 5. \((vts_0, \ldots, vts_{n-1}) := VTS.scan\) |
| 6. for \(j = 0\) to \(n - 1\) do |
| 7. \(vts_i[j] := newts(vts_j[i], vts_i[j])\) |
| 8. VTS.update(vts_i) |
| 9. viewsum := v_0.counter + \cdots + v_{n-1}.counter |
| 10. myview[color] := viewsum |
| 11. myview[color + 1 (mod 3)] := null |
| 12. ViewSum.update(myview) |
| 13. \((view_0, \ldots, view_{n-1}) := ViewSum.scan\) |
| 14. classify(view_0, \ldots, view_{n-1}) |
| 15. flag := newflag() |
| 16. Flags.update(flag) |

| Fscan() |
|----------------------------------|
| 1. \((flag_0, \ldots, flag_{n-1}) := Flags.scan\) |
| 2. winner := find\_max(flag_0, \ldots, flag_{n-1}) |
| 3. return flag.winner.ans |

Figure 2: code for \(p_i\)

The algorithm use 4 local procedures:
1. **classify** - gets \( n \) triples of natural numbers as arguments. Each triple represents the amount of knowledge that the corresponding process has obtained in its recent update operations. As each update operation has a color from the set \( \{0, 1, 2\} \), each entry is a triple in correspondence to three possible colors. This procedure constructs the sets \( \text{flag.winners} \) and \( \text{flag.losers} \) based on the considerations explained above.

2. **newflag**. Creates a new flag before updating the *Flags* object.

3. **newts** - gets two pairs of timestamps: \( \text{pair}_1, \text{pair}_2 \) and returns a pair of timestamps, \( \text{pair}_3 \) such that \( \text{pair}_3.\text{new} \) dominates both fields of \( \text{pair}_1 \), and \( \text{pair}_3.\text{old} = \text{pair}_2.\text{new} \)

4. **find_max** - gets \( n \) flags as arguments and returns an element from \( \{0, \ldots, n-1\} \). This procedure is invoked during an *Fscan* operation and the element that this procedure returns is the id of the most up-to-date process.

The procedures **classify**, **newflag** and **newts** are presented in Figure 3. The procedure **find_max** is discussed in the next subsection.

| classify\((\text{view}_0, \ldots, \text{view}_{n-1})\) | newflag() | newts\((u, v), (u', v')\) |
|---------------------------------|----------|-----------------|
| 1. \( \text{winners} := \) &lt; j &gt; &lt;\text{viewsum}\rangle \cup \{j, c : \text{view}_j[c] = \text{viewsum} \wedge i &lt; j\} &lt; j &gt; &lt;\text{viewsum}\rangle \cup \{j, c : \text{view}_j[c] = \text{viewsum} \wedge i &gt; j\} &lt; j &gt; &lt;\text{viewsum}\rangle \cup \{j, c : \text{view}_j[c] = \text{viewsum} \wedge i &lt; j\} &lt; j &gt; &lt;\text{viewsum}\rangle \cup \{j, c : \text{view}_j[c] = \text{viewsum} \wedge i &gt; j\} | 1. \( \text{flag.color} := \text{color} \) &lt; j &gt; &lt;\text{viewsum}\rangle \cup \{j, c : \text{view}_j[c] = \text{viewsum} \wedge i &lt; j\} &lt; j &gt; &lt;\text{viewsum}\rangle \cup \{j, c : \text{view}_j[c] = \text{viewsum} \wedge i &gt; j\} &lt; j &gt; &lt;\text{viewsum}\rangle \cup \{j, c : \text{view}_j[c] = \text{viewsum} \wedge i &lt; j\} &lt; j &gt; &lt;\text{viewsum}\rangle \cup \{j, c : \text{view}_j[c] = \text{viewsum} \wedge i &gt; j\} | 1. \( \text{ts.old} := u' \) &lt; j &gt; &lt;\text{viewsum}\rangle \cup \{j, c : \text{view}_j[c] = \text{viewsum} \wedge i &lt; j\} &lt; j &gt; &lt;\text{viewsum}\rangle \cup \{j, c : \text{view}_j[c] = \text{viewsum} \wedge i &gt; j\} &lt; j &gt; &lt;\text{viewsum}\rangle \cup \{j, c : \text{view}_j[c] = \text{viewsum} \wedge i &lt; j\} &lt; j &gt; &lt;\text{viewsum}\rangle \cup \{j, c : \text{view}_j[c] = \text{viewsum} \wedge i &gt; j\} | 2. \( \text{ts.new} = (\text{next}(u, v)) \) &lt; j &gt; &lt;\text{viewsum}\rangle \cup \{j, c : \text{view}_j[c] = \text{viewsum} \wedge i &lt; j\} &lt; j &gt; &lt;\text{viewsum}\rangle \cup \{j, c : \text{view}_j[c] = \text{viewsum} \wedge i &gt; j\} &lt; j &gt; &lt;\text{viewsum}\rangle \cup \{j, c : \text{view}_j[c] = \text{viewsum} \wedge i &lt; j\} &lt; j &gt; &lt;\text{viewsum}\rangle \cup \{j, c : \text{view}_j[c] = \text{viewsum} \wedge i &gt; j\} | 3. \( \text{return } \text{ts.old, ts.new} \) &lt; j &gt; &lt;\text{viewsum}\rangle \cup \{j, c : \text{view}_j[c] = \text{viewsum} \wedge i &lt; j\} &lt; j &gt; &lt;\text{viewsum}\rangle \cup \{j, c : \text{view}_j[c] = \text{viewsum} \wedge i &gt; j\} &lt; j &gt; &lt;\text{viewsum}\rangle \cup \{j, c : \text{view}_j[c] = \text{viewsum} \wedge i &lt; j\} &lt; j &gt; &lt;\text{viewsum}\rangle \cup \{j, c : \text{view}_j[c] = \text{viewsum} \wedge i &gt; j\} |

**Figure 3**: Local procedures

### 2.5 The Procedure **find_max**

This procedure is invoked during an execution of an *Fscan* event \( S \), and it returns the id of the most up-to-date process. Thus, the process that executes \( S \) returns the value \( \text{flag.ans} \) in case that **find_max** returns \( i \).

The **find_max** procedure of an *Fscan* operation \( S \) gets \( n \) flags as arguments: \( \text{flags}(S) := (\text{flags}_0, \ldots, \text{flags}_{n-1}) \). The procedure returns a maximal element in relation \( \text{<}_S \subseteq \{0, \ldots, n-1\} \times \{0, \ldots, n-1\} \) that we define here. The relation \( \text{<}_S \) is defined by reference to \( \text{flags}(S) \) in definition 1.

**Definition 1.** Let \( p_i \) and \( p_j \) be two processes and write: \( \text{flag}_i.\text{color} = c_i \) and \( \text{flag}_j.\text{color} = c_j \). We say that \( p_i \) and \( p_j \) are in conflict, if one of the following occurs:

1. \( (j, c_j) \in \text{flag}_i.\text{winners} \) and \( (i, c_i) \in \text{flag}_j.\text{winners} \).

2. \( (j, c_j) \in \text{flag}_i.\text{losers} \) and \( (i, c_i) \in \text{flag}_j.\text{losers} \).

Definition 1 is important since, as we shall prove, for each two processes \( p_i \) and \( p_j \) and an *Fscan* event \( S \), the *flag* of one of these processes determines correctly the ordering between \( p_i \) and \( p_j \). That is, if \( p_i \) is the reliable process and if (for example) \( (j, c_j) \in \text{flag}_i.\text{winners} \) and the color in \( p_j \)'s *flag* is \( c_j \), then \( p_j \) is indeed more up-to-date than \( p_i \) (more precisely, the *ans* field of \( p_j \)'s *flag* is more up-to-date) as indicated by \( p_i \)'s *flag*. The problem is that we do not know which process provides correct information among any pair of processes. However, this problem does not arise when the processes are not in conflict. When processes provide contradicting information we use the processes’ timestamps to find the trustworthy process.
Definition 2. Let \( p_i \) and \( p_j \) be two processes and write: \( \text{flag}.\text{color} = c_i, \text{flag}.\text{color} = c_j \). \( i < S j \) if one of the following occurs:

1. \( p_i \) and \( p_j \) are not in conflict and \((i, c_i) \in \text{flag}.\text{losers}\).
2. \( p_i \) and \( p_j \) are not in conflict and \((j, c_j) \in \text{flag}.\text{winners}\).
3. \( p_i \) and \( p_j \) in conflict, \( \text{flag}.\text{vts}[j].\text{new} <_S \text{flag}.\text{vts}[i].\text{new} \) and \((i, c_i) \in \text{flag}.\text{losers}\).
4. \( p_i \) and \( p_j \) in conflict, \( \text{flag}.\text{vts}[i].\text{new} <_S \text{flag}.\text{vts}[j].\text{new} \) and \((j, c_j) \in \text{flag}.\text{winners}\).

An element \( i \in \{0, \ldots, n - 1\} \) is maximal in \( <_S \) if there is no \( j \neq i \) such that \( i <_S j \). The procedure \( \text{find}_{\text{max}}(\text{flag}_0, \ldots, \text{flag}_{n-1}) \) (line 2) returns a maximal element in \( <_S \) (we shall prove that such a maximal element exists in any \( \text{Fscan} \) event). This procedure \( \text{find}_{\text{max}} \) accesses only local variables and we omit the technical but easy implementation of this procedure.

3 Correctness

Fixing an execution \( \tau \), we need to show that the precedence relation defined over the high level events in \( \tau \), \( < \) can be extended into a linear ordering, \( \prec \) that belongs to the sequential specification of the \( F \)-snapshot object. In section 2.4 a relation \( <_S \) was defined. An \( \text{Fscan} \) event \( S \) returns a value stored in \( \text{Flags}[j].\text{ans} \) where \( j \) is maximal in relation \( <_S \). Thus, for proving correctness we also need to show that for any \( \text{Fscan} \) event in \( \tau \), \( S \), \( <_S \) admits a maximal element.

Since the algorithm is wait-free, we may assume that all operations in \( \tau \) are complete. Indeed, if there are pending operations in \( \tau \), we can let the processes take additional steps and complete the pending operation. This way, an execution that extends \( \tau \) is obtained. A linearization of the resulting execution admits a linearization of \( \tau \) as well.

As explained in the preliminaries section, at the beginning of \( \tau \), each process performs an initialization and writes initial values to the registers and variables. These initial high level events precede all other actions in \( \tau \) and are considered as \( \text{update} \) events. If \( I_j \) is such an initial event by \( p_j \), the value of this event, \( \text{val}(I_j) \) is \( x_0 \). Recall that the initial value of the entry \( \text{Vals}[j] \) is \((0, x_0)\).

The execution \( \tau \) is a sequence of atomic actions addressed to the shared memory. Thus, the procedure executions addressed to the snapshot objects (e.g line 2 of the \( \text{update} \) procedure) are not atomic and represent a sequence of actions that a process executes. However, by using a linearizable implementation for the snapshot objects, we may assume for convenience that all the procedure executions addressed to the snapshot objects are atomic. This assumption simplifies our proof since we do not need to speak about the linearization points of these operations and the corresponding extension of \( < \). For further discussion about using linearizable implementations see \([9]\) and \([17]\).

Our algorithm employs several snapshot objects. Thus, for preventing confusion, we use the notation \( A.\text{update} \) and \( A.\text{scan} \) to denote invocations of \( \text{update} \) and \( \text{scan} \) procedures addressed to object \( A \). Note that an \( A.\text{update}(x) \) invocation by \( p_i \) writes \( x \) to the \( i \)-th segment of \( A \).

If \( e \) is a read (write) event executed by some of the processes, we use \( \text{val}(e) \) to denote the value that the process read (wrote) in \( e \). Similarly, if \( e \) is an \( A.\text{update} \) event addressed to a snapshot object \( A \), \( \text{val}(e) \) is the value that the executing process wrote to the corresponding segment of \( A \), and if \( e \) is an \( A.\text{scan} \) event, \( \text{val}(e) \) is the vector of elements that \( e \) returns. Any low level event \( e \) belongs to a unique high level event, which is an \( \text{update} \) or an \( \text{Fscan} \) event by some of the processes. We use \([e]\) to denote this event. It is clear that \( e \in [e] \).

The following notations are important in our proof:

1. For an \( A.\text{scan} \) event \( e \) on a snapshot object \( A \), we define \( \mu_j(e) \) to be the maximal \( A.\text{update} \) event by \( p_j \) that precedes \( e \). Thus, \( \text{val}(e)[j] = \text{val}(\mu_j(e)) \) for any \( A.\text{scan} \) event \( e \).
2. Let \( U \) be an update event and \( p_i \) a process. If \( U \) is an initial update event we set \( \alpha_i(U) = I_i \), the initial \( p_i \)-update event. Otherwise, \( \alpha_i(U) \) is the \( p_i \)-update event in which \( p_i \) wrote to \( V[i] \) the value that was read from \( V[i] \) in \( U \). That is:
\[
\alpha_i(U) = [\mu_i(V.\text{scan}(U))]
\]
where \( V.\text{scan}(U) \) is the (unique) \( V.\text{scan} \) event in \( U \), which corresponds to the execution of line 3 in the code of the update procedure.

3. Let \( S \) be an Fscan event in which \( \text{winner} = j \) (the invocation of find\_max in \( S \) returns \( j \)). Let \( e \) be the (unique) Flags.\text{scan} event in \( S \), and let \( U_j = [\mu_j(e)] \). For a process id \( i \), we define \( \alpha_i(S) = \alpha_i(U_j) \).

4. Let \( S \) be an Fscan event. For a process id \( i \), \( \beta_i(S) \) is the \( p_i \)-update event that wrote to Flags\[i the value read in \( S \). That is, \( \beta_i(S) = [\mu_i(Flags.\text{scan}(S))] \) where Flags.\text{scan}(S) is the (unique) Flags.\text{scan} event in \( S \).

Two easy observations that will be useful later are the following:

**Lemma 3.** For each \( p_i \)-update event \( U \), \( \alpha_i(U) = U \).

**Lemma 4.** Let \( U_1 \) and \( U_2 \) be two update events such that \( U_1 < U_2 \). Then, for each process id \( i \), \( \alpha_i(U_1) \leq \alpha_i(U_2) \).

Lemma 3 holds since the V.update operation in \( U \) precedes the V.scan operation (lines 2 and 3). Lemma 4 holds since the V.scan event in \( U_1 \) precedes the V.scan event in \( U_2 \).

We fix a compete Fscan event \( S \) and we shall prove that there is a maximal element in relation \( <_S \). For each process id \( i \), write \( U_i = \beta_i(S) \), \( \text{flag}_i = \text{val}(\mu_i(Flags.\text{scan}(S))) \), and \( c_i = \text{flag}_i.\text{color} \). That is, \( U_i \) is the \( p_i \)-update event that wrote to Flags the value read in \( S \), \( \text{flag}_i \) is the value that \( p_i \) wrote to Flags\[i in \( U_i \) and \( c_i \) is the value of the color field of \( \text{flag}_i \). According the initial values of the snapshot object Flags, and by the code of the classify procedure, the following hold:

**Lemma 5.** For a pair \( (j, c) \in \{0, \ldots, n-1\} \times \{0, 1, 2\} \) and a process \( p_i \), at most one of the following occurs:

1. \( (j, c) \in \text{flag}_i.\text{winners} \).
2. \( (j, c) \in \text{flag}_i.\text{losers} \).

**Corollary 6.** If \( i <_S j \), then \( \neg(j <_S i) \).

**Proof.** Consider definition 2 and observe that relation \( <_S \) between \( i \) and \( j \) is determined only by the flag of one of these processes. Hence, this is a consequence from the previous lemma.

If \( U_i \) is not the initial update event \( I_i \), when \( p_i \) executed \( U_i \), it computed a natural number while executing line 9. Let \( m_i \) denote this number. If \( U_i = I_i \), define \( m_i = 0 \). We argue that \( m_i \) reflects correctly how recent \( p_i \)'s view is.

**Lemma 7.** For two processes \( p_i \) and \( p_j \), if \( (m_i, i) < (m_j, j) \) at the lexicographic order, then for each process id \( k \), \( \alpha_k(U_i) \leq \alpha_k(U_j) \).

**Proof.** If \( U_i = I_i \), then for each process id \( k \), \( \alpha_k(U_i) \) is the first \( p_k \)-update event and the lemma hold. If \( U_j = I_j \), then \( m_j = 0 \) which implies that \( m_i = 0 \). Thus, \( U_i = I_i \) and we are done. It is left to deal with the case that \( U_i \neq I_i \) and \( U_j \neq I_j \).

Towards a contradiction, assume that \( \alpha_k(U_j) < \alpha_k(U_i) \) for some process id \( k \). We conclude that the V.scan operation in \( U_i \) occurred after the V.scan operation in \( U_j \). Therefore, the counter that \( p_i \) read from each field \( V[l].\text{counter} \) is larger than the counter that \( p_j \) read (note that the \( l \)-th update operation by each process writes \( l \) to this field). Hence, \( m_j \leq m_i \). However, since the integer that \( p_i \) read from \( V[k].\text{counter} \) is strictly larger than the one that \( p_j \) read, \( m_j < m_i \) in contradiction to the assumption that \( (m_i, i) < (m_j, j) \).
During the execution of \( S \), for each two processes \( p_i \) and \( p_j \), the process that executes \( S \) decides whether \( i <_S j \) or \( j <_S i \). The decision is made upon the values of these processes’ flags. The next lemmas show that for each such a pair of processes, at least one of these processes’ flags provides reliable information. That is to say, for some process (say, \( p_i \)) the following occurs:

- If \( \text{flag}_i.\text{ans} \) is more up-to-date than \( \text{flag}_j.\text{ans} \), then \( (j, c_j) \in \text{flag}_i.\text{losers} \).
- If \( \text{flag}_i.\text{ans} \) is less up-to-date than \( \text{flag}_j.\text{ans} \), then \( (j, c_j) \in \text{flag}_i.\text{winners} \).

**Lemma 8.** Let \( p_i \) and \( p_j \) be two processes such that \( U_i \neq I_i \). Let \( e_i \in U_i \) be the update of ViewSum in \( U_i \) (line 12) and let \( e_j \) be the update of ViewSum in \( U_j \). Let \( e \) be the scan event of ViewSum in \( U_i \) (line 13). If \( e_j < e_i \), then one of the following holds:

1. \( \mu_j(e) = e_j \) or,
2. \( \mu_j(e) = e' > e_j \) and there is no \( p_j \)-update event between \( U_j = [e_j] \) and \( [e'] \).

**Proof.** Since \( e_j < e_i \) and since (by the code) \( e_i < e \), we see that \( e_j < e \) and hence, \( e_j \leq \mu_j(e) \). Thus, we need to show that there is at most one ViewSum.update event by \( p_j \) between \( e_j \) and \( e \).

Assume for a contradiction that \( e' \) and \( e'' \) are two ViewSum.update events by \( p_j \) such that

\[
e_j < e' < e'' < e.
\]

Each ViewSum.update event belongs to a unique update event so there are two different \( p_j \)-update operations \( U' = [e'] \) and \( U'' = [e''] \). Recall that \( [e_j] = U_j \) and observe that:

\[
\beta_j(S) = U_j < U' < e'' < e.
\]

Now, the Flags.scan event in \( S \) occurs after \( e \), so it reads the value written to Flags[j] in \( U' \) or in a later event. We have:

\[
\beta_j(S) = U_j < U' \leq [\mu_j(\text{Flags.update}(S))]
\]

in contradiction to the definition of \( \beta_j(S) \).

We conclude:

**Lemma 9.** Let \( p_i \) and \( p_j \) be two processes such that \( U_i \neq I_i \). Let \( e_i \in U_i \) be the update of ViewSum in \( U_i \), let \( e_j \) be the update of ViewSum in \( U_j \), and let \( e \) be the scan event of Views in \( U_i \). If \( e_j < e_i \), then \( p_i \) reads \( m_j \) from ViewSum[j][c_j] in \( e \), and in addition:

1. If \( (m_i, i) < (m_j, j) \) (at the lexicographic order), then \( (j, c_j) \in \text{flag}_i.\text{winners} \).
2. If \( (m_i, i) > (m_j, j) \) (at the lexicographic order), then \( (j, c_j) \in \text{flag}_i.\text{losers} \).

**Proof.** \( e \in U_i \) is the ViewSum.scan event in \( U_i \) by \( p_i \). By the previous lemma, since \( e_j < e_i \) there is at most one write to ViewSum between \( e_j \) and \( e \). Thus, the value that \( p_j \) wrote to ViewSum[j][c_j] (which is \( m_j \)) has not been “deleted” (consider lines 10-12 in the update procedure). \( p_i \) reads \( m_j \) from ViewSum[j][c_j] and the lemma follows from the code of the classify procedure and from lemma 8.

So far, we have proved that for any two processes \( p_i \) and \( p_j \), one of these processes’ flags provides reliable information. Namely, the process that wrote later to ViewSum during the update events \( U_i \) and \( U_j \). The next lemma easily stems.

**Lemma 10.** Let \( p_i \) and \( p_j \) be two processes which are not in conflict (the definition is in section 2.6). If \( (m_i, i) < (m_j, j) \) lexicographically, then \( i <_S j \).
Proof. First assume that \( m_j = 0 \). In this case, \( U_j = I_j \) the initial update event. In addition, since \((m_i, i) < (m_j, j)\), also \( m_i = 0 \) thus \( U_i = I_i \) as well. We conclude that \( i < j \) and according to the initial values of the registers we get that \((j, 0) \in Flags[i].\text{winners}\) and \((i, 0) \in Flags[j].\text{losers}\). In addition, \( c_i = c_j = 0 \) and hence, \( i < j \) as required.

Now assume that \( m_j > 0 \) and conclude that \( U_j \neq I_j \). Write \( e_i \in U_i \) - the update of ViewSum in \( U_i \) and respectively, \( e_j \) is the update of ViewSum in \( U_j \). Assume w.l.o.g. that \( e_j < e_i \) and observe that \( U_i \) is not the initial event either. By lemma 9 \((j, c_j) \in flag_i.\text{winners}\). Since the processes are not in conflict the claim holds.

Our next goal is to prove the same for the case that the processes are in conflict. If the processes are in conflict, we know by the previous lemmas that one of them provides reliable information. Recall that in this case, the definition of \( <_S \) is according to the flag of the process that presents a later timestamp. We need to show that the process with the later timestamp is the reliable one, namely the one that wrote later to ViewSum.

**Lemma 11.** Let \( p_i \) and \( p_j \) be two processes. Let \( e_i \in U_i \) be the update of ViewSum in \( U_i \), let \( e_j \) be the update of ViewSum in \( U_j \), and assume that \( e_j < e_i \). If \( p_i \) and \( p_j \) are in conflict, then \( flag_i.vts[i].\text{new} <_t S flag_j.vts[j].\text{new} \).

**Proof.** First, note that \( U_i \neq I_i \). Indeed, if \( U_i = I_i \) we get that also \( U_j = I_j \) (since \( e_j < e_i \)) which implies that the processes are not in conflict.

For the rest of the proof we assume that \( U_j \neq I_j \). If \( U_j = I_j \), then similar (and simpler) argument can be applied. Let \( s_j \) be the scan of ViewSum in \( U_j \). By lemma 8 \( p_i \) reads \( m_j \) from \( ViewSum[j][c_j] \) in \( U_i \), but since \( p_i \) and \( p_j \) are in conflict, conclude that \( p_j \) read some \( k \neq m_i \) from \( ViewSum[i][c_i] \) in \( s_j \). Hence,

\[
\mu_i(s_j) \neq e_i.
\]

Since \( e_j < e_i \) and (by the code) \( e_j < s_j \), either \( e_j < s_j < e_i \) or \( e_j < e_i < s_j \). We claim that the first option occurs and \( s_j < e_i \). Assume otherwise and use equation 1 to conclude that \( e_i < \mu_i(s_j) \). Note that there can be at most one ViewSum.update event by \( p_i \) that follows \( e_i \) and precedes \( s_j \) (consider the arguments in the proof of lemma 8), and hence \( s_j \) reads from ViewSum[i] the value of this event. However, by the code of the update procedure, the update operation by \( p_i \) that follows \( U_i \) also writes \( m_i \) to ViewSum[i][c_i]. Thus, if \( e_i < s_j \), then \( p_j \) reads \( m_i \) from ViewSum[i][c_i] in \( s_j \), and this is in contradiction to the assumption that the processes are in conflict.

Now we claim that there is a \( p_i.\text{ViewSum.update} \) event between \( s_j \) and \( e_i \). Indeed, assume not and let \( e' \) be the last ViewSum.update event by \( p_i \) that precedes \( e_i \). By our assumption we have \( \mu_i(s_j) = e' \). e' belongs to the last \( p_i.\text{update} \) event that precedes \( U_i \) and hence the color of this update event is \( c_i - 1 \) (mod 3). Therefore, \( val(e')[c_i] = \text{null} \). We conclude that \( p_j \) read null from ViewSum[i][c_i] in \( s_j \) and this contradicts the fact that \( p_i \) and \( p_j \) are in conflict.

We see that there is a ViewSum.update event by \( p_i \) between \( s_j \) and \( e_i \), say \( e'_i \). That is,

\[
s_j < e'_i < e_i.
\]

Write \([e'_i] = U'\), a \( p_i.\text{update} \) event and note that \( U' < U_i \). Therefore, \( e'_i < U_i \) and hence

\[
s_j < U_i.
\]

Now, let \( t_j \in U_j \) be the (unique) VTS.update event in \( U_j \) (line 8 in the code) and write \( \text{val}(t_j)[i] = (x, y) \). Observe that since \( t_j \in U_j \), \((x, y)\) is also the value of \( flag_j.vts[i] \). Let \( s_i \in U_i \) be the (unique) VTS.scan event in \( U_i \) (line 5) and note that since \( e'_i < U_i \), \( e'_i < s_i \). By the code and by our conclusions we have:

\[
t_j < s_j < e'_i < s_i \text{ thus } \mu_j(s_i) \geq t_j.
\]

Note that there is at most one VTS.update event by \( p_j \) between \( t_j \) and \( s_i \) since otherwise, we would have \( \beta_j(S) \neq U_j \). Furthermore, if there is such an event, it writes to VTS[j][i]: \((y, z)\) for some vertex \( z \in V_S \) (consider the newts code). Let \((a, b)\) denotes the value of VTS[i][j] before the execution of \( U_i \).
Case 1. \( \mu_j(s_i) = t_j \) and hence \( p_i \) reads in \( s_i \) from \( \text{VTS}[j][i] \): \((x, y)\). Thus, \( p_i \) writes in \( U_i \) to \( \text{Flags}[i].\text{vts}[j] : \text{newts}(x, y), (a, b) = (b, \text{next}(x, y)) \). Since \((\text{next}(x, y), y) \in E_G \), \( \text{flag}_j.\text{vts}[i].\text{new} < _{ts} \text{flag}_i.\text{vts}[j].\text{new} \) as required.

Case 2. \( \mu_j(s_i) > t_j \) and hence \( p_i \) reads in \( s_i \) from \( \text{VTS}[j][i] \): \((y, z)\). In this case \( p_i \) writes in \( U_i \) to \( \text{Flags}[i].\text{vts}[j] : \text{newts}(y, z), (a, b) = (b, \text{next}(y, z)) \). Since also \((\text{next}(y, z), y) \in E_G \), \( \text{flag}_j.\text{vts}[i].\text{new} < _{ts} \text{flag}_i.\text{vts}[j].\text{new} \).

We see that the lemma holds in this case as well.

The previous lemma shows that if two processes are in conflict and their \textit{flags} provide contradicting information, the \textit{flag.vts} fields determined correctly which among the two processes is the reliable one. The conclusion is that relation \(<_S\) determines correctly which process presents the most up-to-date view in its \textit{flag.ans} field.

**Lemma 12.** Let \( p_i \) and \( p_j \) be two processes. Then, \((m_i, i) < (m_j, j) \iff i <_S j\).

**Proof.** First assume that \((m_i, i) < (m_j, j)\) and we shall prove that \( i <_S j \). If \( p_i \) and \( p_j \) are not in conflict, then this is the case of lemma \[12\]. If \( p_i \) and \( p_j \) are in conflict, let \( e_i \) be the update of ViewSum in \( U_i \) and let \( e_j \) be the update of ViewSum in \( U_j \). Assume w.l.o.g. that \( e_j < e_i \). By lemma \[12\] \((j, c_j) \in \text{flag}_i.\text{winners}\).

Now, for the other direction, assume that \( i <_S j \). If \((m_j, j) < (m_i, i)\), then we get that also \( j <_S i \) in contradiction to corollary \[12\]. Thus, \((m_i, i) < (m_j, j)\) as required.

It is easy to see that any two update events \( U \) and \( U' \), are comparable in \( \leq_a \). For verifying this observation, assume w.l.o.g. that the \textit{V.scan} event in \( U' \) occurs after the \textit{V.scan} event in \( U \). Clearly, \( U \leq_a U' \) in this case. Therefore, we conclude:

**Corollary 13.** There is a maximal element in \(<_S\) and hence, the \textit{find.max} procedure in \( S \) returns some \( j < n \).

**Proof.** Take the (unique) \( j \) such that \((m_j, j)\) is maximal at the lexicographic order over \\{\((m_0, 0), (m_1, 1), \ldots, (m_{n-1}, n-1)\)\}. By the previous lemma, \( j \) is maximal in relation \( i <_S j \).

Now we are ready to show that \( \tau \) is a linearizable. We define a linear ordering \( < \) on the set of all high-level events in \( \tau \). First, we define \( < \) over the update events. Then, we define \( < \) between update and Fscan events and finally, we define \( < \) over Fscan events.

1. For two update operations \( U, U' \), we set \( U < U' \) if the write to \( V \) in \( U \) precedes the write to \( V \) in \( U' \).

That is, the executions of the V.update operations are the linearization points of the update events.

2. Let \( S \) be an Fscan event. For each update event \( U \), we decide if \( U < S \) or \( S < U \) by choosing an update event to linearize \( S \) immediately after it.

For each process \( p_i \), write \( U_i = \alpha_i(S) \). We linearize \( S \) immediately after the initial update events \( U_0, \ldots, U_{n-1} \). More precisely, we linearize \( S \) after the maximal element in \( < \) over the set \{\( U_0, \ldots, U_{n-1} \)\}.

3. It is left to define \( < \) over the Fscan events. First, we consider pairs of Fscan events \( S, S' \) such that \( S \) was linearized after an update event \( U \) and \( S' \) was linearized after an update event \( U' \neq U \). In this case, if \( U < U' \), we set \( S < S' \).

Now, for each update event \( U \), we take the Fscan events linearized after \( U, S_1, \ldots, S_m \), and we extend \( < \) on these events in some arbitrary way that extends \( < \) over \( S_1, \ldots, S_m \).

It is easy to verify that \( < \) is a linear ordering, now we verify that \( < \) extends \( < \). Consider two high-level events \( A < B \). We shall prove that \( A < B \). The claim is trivial when \( A \) and \( B \) are update events as these events were linearized in correspondence to an execution of an atomic instruction. We need to deal with the cases that \( A \) and \( B \) are both Fscan events, or one of them is an Fscan event and the other is an update event.
Case 1. \( A = U \) an update event, say by \( p_i \) and \( B = S \) an Fscan event. For each process \( p_k \), let \( U_k \) denote the \( p_k \)-update event that wrote to Flags\([k]\) the value read in \( S \). i.e. \( U_k = \beta_k(S) \). Note that \( U \leq U_i \). Assume that the procedure \texttt{find\_max} in \( S \) returned \( j \) thus \( i \leq S j \) and \( \alpha_i(S) = \alpha_i(U_j) \). Use lemmas 12 and 3 to observe that:

\[
U \leq U_i = \alpha_i(U_i) \leq \alpha_i(U_j) = \alpha_i(S).
\]

Recall that \( S \) was linearized after \( \alpha_i(S) \) and hence, since \( \prec \) extends \( \prec \) over update events,

\[
U \leq U_i = \alpha_i(U_i) \leq \alpha_i(U_j) \prec S
\]

which implies that \( U \prec S \).

Case 2. \( A = S \) an Fscan event and \( B = U \) an update event, say by \( p_i \). Note that since \( S < U \), \( U \neq I_i \) the initial \( p_i \)-update event. To show that \( S \) was linearized before \( U \), we need to show that for each process \( p_k \), \( \alpha_k(S) \prec U \).

Assume that the procedure \texttt{find\_max} in \( S \) returns \( j \) and write \( U_j = \beta_j(S) \). For a process \( p_k \), write \( U_k = \alpha_k(S) = \alpha_k(U_j) \). If \( U_j = I_j \), then \( U_k = I_k \) and then it is clear that \( U_k \prec U \) as required. Otherwise, let \( e_k \) be the write to \( V \) in \( U_k \), let \( r \) be the V.scan event in \( U_j \) and let \( e \) be the V.update event in \( U \). Obviously, \( e_k < r \). Since \( \beta_j(S) = U_j \prec (S < r) \). But since \( S < U \), we conclude that \( r < e \).

As a result, \( e_k < e \) which implies that \( U_k \prec U \) as required.

Case 3. \( A = S \) and \( B = S' \) are both Fscan event. For proving that \( S \) is linearized before \( S' \) we show that for each process \( p_i \), \( \alpha_i(S) \leq \alpha_i(S') \). Assume that the procedure \texttt{find\_max} in \( S \) returns \( j \) and the procedure \texttt{find\_max} in \( S' \) returns \( k \). Write \( U_j = \beta_j(S) \) and \( U_k = \beta_k(S') \). Hence, \( \alpha_i(S) = \alpha_i(U_j) \) and \( \alpha_i(S') = \alpha_i(U_k') \). Write \( U_j' = \beta_j(S') \) and use lemmas 12 and 7 to conclude that \( \alpha_i(U_j') \leq \alpha_i(U_k') \).

Since \( S < S' \), \( U_j \leq U_j' \) thus by lemma 4 we get,

\[
\alpha_i(S) = \alpha_i(U_j) \leq \alpha_i(U_j') \leq \alpha_i(U_k) = \alpha_i(S).
\]

Hence \( \alpha_i(S) \leq \alpha_i(S') \) and \( \alpha_i(S) < \alpha_i(S') \) follows.

It is left to prove that the properties of the sequential specification are satisfied. It is easy to see that each Fscan event \( S \) returns \( F(val(\alpha_0(S)), \ldots, val(\alpha_{n-1}(S))) \). Therefore, we need to verify that for each process \( p_i \), \( \alpha_i(S) \) is the maximal \( p_i \)-update event that precedes \( S \) in \( \prec \). Since \( S \) was linearized after the events \( \alpha_0(S), \ldots, \alpha_{n-1}(S) \), clearly \( \alpha_i(S) \prec S \) for each process \( p_i \).

Towards a contradiction, assume that for some process \( p_i \), \( U \neq \alpha_i(S) \) is the maximal \( p_i \)-update event that precedes \( S \) in \( \prec \). Hence,

\[
\alpha_i(S) \prec U \prec S.
\]

We conclude that there is a process \( p_k \) such that

\[
\alpha_i(S) \prec U \prec \alpha_k(S)
\]

since otherwise, \( S \) would have linearized before \( U \). Note that \( \alpha_k(S) \neq I_k \). Assume that the procedure \texttt{find\_max} in \( S \) returns \( j \) and write \( U_j = \beta_j(S) \). Thus, \( \alpha_k(S) = \alpha_k(U_j) \). Since \( \alpha_k(S) \) is not the initial \( p_k \)-event, also \( U_j \neq I_j \).

Write \( \alpha_i(S) = \alpha_i(U_j) = U_i \) and \( \alpha_k(S) = \alpha_k(U_j) = U_k \). Let \( e_i \) be the V.update operation in \( U_i \), let \( e \) be the V.update operation in \( U \) and let \( e_k \) be the V.update operation in \( U_k \). Since \( U_i \prec U \prec U_k \), we have

\[
e_i < e < e_k.
\]

Now, let \( r \) be the V.scan event in \( U_j \). By definition, \( \mu_k(r) \in U_k \) thus \( \mu_k(r) = e_k \) and in particular

\[
e_k < r.
\]

As a result, \( e_i < e < r \) thus \( \mu_i(r) \neq e_i \) in contradiction to \( \alpha_i(U_j) = U_i \).
4 Conclusions

The snapshot object is a special case of the $F$-snapshot object while choosing the parameter $F$ to be the identity function. The $F$-snapshot object also generalizes the signaling object $\text{seq}$ from the case that there only two processes to an arbitrary number of processes. We present here a wait free solution to this problem. Our algorithm uses several snapshot objects thus its complexity measures depend on the exact implementations of this objects.

When processes communicate through shared read/write registers, any $\text{Fscan}$ implementation must include $\Omega(n)$ operations addressed to the shared memory [25]. Furthermore, regarding the snapshot object, Israeli and Shirazi [21] proved the same lower bound for $\text{update}$ implementations. As the snapshot object is a special case of the $F$-snapshot object, this lower bound holds for the $F$-snapshot object as well.

The exact implementations for the snapshot objects that are used in our algorithm, determine the time complexity of the algorithm. However, since the $\text{Flags}$ object is accessed during $\text{Fscan}$ operations, it is required to use a snapshot implementation that employs only bounded registers, in case that only finitely many different values are invoked by the processes. As an example, the first algorithm in [3] violates this requirement since it uses a field named $\text{seq}$ that grows infinitely, while the second algorithm in [3] satisfies this property.

For efficiency, we can use the implementation by Attiya and Rachman in [11]. In section 4.4 of [11], the authors explain how to transform their algorithm into a snapshot implementation which satisfies the requirements discussed here. Namely, into a snapshot implementation that uses only bounded registers, in case that finitely many data values may be written to the segments of the object. Thus, our algorithm can be implemented with time complexity $O(n \log n)$ which is, as far as we know, the time complexity of the most efficient published snapshot algorithm that uses only single-writer registers.

It is known that the snapshot object can be implemented with time complexity $O(n)$ when multi-writers are allowed as Inoue, Masuzawa, Chen and Tokura proved [18]. Inoue et al. present an algorithm that solves the lattice agreement problem. Then, the reduction by Attiya, Herlihy and Rachman [10], provides a linear $\text{Fscan}$ implementation using multi-writer registers. The problem is that this reduction requires unbounded memory. Hence, the $F$-snapshot limitations forbid using this implementation for the $\text{Flags}$ snapshot object in our algorithm. Therefore, the question if there is a linear $F$-snapshot implementation using multi-writers is not answered here, although there is a linear snapshot implementation that uses multi-writer registers.

By the essence of the problem, a natural complexity measure for an $F$-snapshot implementation is the size of the flags - the bounded registers that are accessed during an $\text{Fscan}$ operation. This “flags complexity” depends on: $n$ - the number of processes and $|D|$ - the number of distinct values that $F$ may return. Since at least $\log |D|$ bits are required to represent $|D|$ different values, it is not difficult to prove that the flags complexity of any solution is $\Omega(n \log |D|)$. For proving this claim, consider an $n$-variable function $F$ that satisfies the following: If we assign values to $n-1$ variables, then any element from $D = \text{rng}(F)$ can be obtained by some assignment to the last variable. For example, the function $f(a_1, \ldots, a_n) = a_1 + \cdots + a_n \pmod{|D|}$ satisfies this requirement. Now, since each process can execute an $\text{update}$ operation and change the result of an ensuing $\text{Fscan}$ event into any element from $D = \text{rng}(F)$, the size of each flag is at least $\log |D|$ bits and the lower bound holds. However, we do not know to prove any non-trivial lower bound on the size of the flags.

For calculating the flags complexity of our algorithm, for convenience, we may assume that $D = \{0, \ldots, |D| - 1\}$. Otherwise, we can take a bijective function $f : D \rightarrow \{0, \ldots, |D| - 1\}$ and replace the function $F$ with $f \circ F$. The $\text{flag}$ type consists of several fields. The field $\text{flag.ans}$ contains elements from $D$ and hence it requires $\log |D|$ bits. The size of the other fields depends only on $n$ - the number of processes. The set fields: $\text{flag.winners}$, $\text{flag.losers}$ can be represented using $3n$ bits, when each bit corresponds to a pair $(i, c)$ of process id and a color. Therefore, all other fields are of size $O(n)$ thus the values that the processes write to $\text{Flags}$ require $O(n + \log |D|)$ bits. However, as $\text{Flags}$ is a snapshot object, the implementation for this object uses additional fields and the largest one contains a view. Namely, it contains a vector of $n$ elements, each entry store a value of the type that the processes write to the snapshot object. Thus, the size of each $\text{flag}$ is actually $O(n^2 + n \log |D|)$ bits. The total size of the flags - which is the flags complexity of the algorithm is $O(n^3 + n^2 \log |D|)$. We believe that this can be significantly improved.
In our algorithm, the \textsc{Fscan} procedure accesses only bounded registers due to the problem constrains and the \textsc{update} procedure access unbounded registers (otherwise the problem is unsolvable). The segments of the snapshot object \(V\) store elements from \(\text{Vals}\) (which might be infinite), and counters that grow infinitely. Hence, if we take a function \(F\) with a finite domain, the \textsc{update} procedure will still access unbounded registers. Thus, in those cases, it is better to use some other implementation such as the bounded version of the algorithm in \cite{11}. An interesting question that arises is whether there is a \(F\)-snapshot algorithm that satisfies both properties:

1. If \(F\) has a finite range, then the \textsc{Fscan} procedure accesses only bounded registers.
2. If \(F\) has a finite domain, then only bounded registers are accessed.

References

\cite{1} U. Abraham and G. Amram. On the Mailbox Problem. In \textit{Principles of Distributed Systems}, pages 453-468, 2014.

\cite{2} Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. In \textit{Proceedings of the 9th Annual Symposium on Principles of Distributed Computing}, pages 1-14, 1990.

\cite{3} Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. \textit{Journal of the ACM}, 40(4):873-890, September 1993.

\cite{4} M. K. Aguilera, E. Gafni, L. Lamport. The mailbox problem (Extended Abstract). In Distributed Computing, pages 1-15, 2008.

\cite{5} M. K. Aguilera, E. Gafni, and L. Lamport. The mailbox problem. \textit{Distributed Computing}, 23(2):113-134, 2010.

\cite{6} R. Alur, K. McMillan, and D. Peled. Model-checking of correctness conditions for concurrent objects. In \textit{Logic in Computer Science (LICS)}, pages 219-228, 1996.

\cite{7} G. Amram. On the signaling problem. In \textit{International Conference on Distributed Computing and networking}, pages 44-65, 2014.

\cite{8} J. Anderson. Composite registers. \textit{Distributed Computing}, 6(3):141-154, 1993.

\cite{9} J. Aspnes and M. Herlihy. Wait-free data structures in the asynchronous PRAM model. In \textit{Proceedings of the 2nd Annual ACM Symposium on Parallel Architectures and Algorithms}, pages 340-349, 1990.

\cite{10} H. Attiya, M. Herlihy, and O. Rachman. Atomic snapshots using lattice agreement. \textit{Distributed Computing}, 8(3):121-132, 1995.

\cite{11} H. Attiya and O. Rachman. Atomic snapshots in \(O(n \log n)\) operations. In \textit{Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing}, pages 29-40, 1993.

\cite{12} D. Dolev and N. Shavit. Bounded concurrent time-stamp systems are constructible. In \textit{Proc. of 21st STOC}, pages 454-466, 1989.

\cite{13} D. Dolev and N. Shavit. Bounded concurrent time-stamping. \textit{SIAM J. Comput.}, 26(2):418-455, 1997.

\cite{14} C. Dwork and O. Waarts. Simple and efficient bounded concurrent timestamping or bounded concurrent timestamp systems are comprehensible!. In \textit{Proceedings of the twenty-fourth annual ACM symposium on Theory of computing}, pages 655-666, 1992.
[15] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745-770, 1993.

[16] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, NY, USA, 2008.

[17] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463-492, 1990.

[18] M. Inoue and W. Chen. Linear-time snapshot using multi-writer multi-reader registers. In *WDAG 94: Proceedings of the 8th International Workshop on Distributed Algorithms*, pages 130-140, 1994.

[19] A. Israeli and M. Li. Bounded time-stamps. *Distributed Computing*, 6(4):205-209, 1993.

[20] A. Israeli, A. Shaham, and A. Shirazi. Linear-time snapshot implementations in unbalanced systems. *Mathematical Systems Theory*, 28(5):469-486, 1995.

[21] A. Israeli and A. Shirazi. The time complexity of updating snapshot memories. *Information Processing Letters*, 65(1):33-40, 1998.

[22] P. Jayanti. f-arrays: Implementation and applications. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, pages 270-279, 2002.

[23] P. Jayanti, K. Tan and S. Toueg. Time and space lower bounds for nonblocking implementations. *SIAM Journal on Computing*, 30(2):438-456, 2000.