Chapter 8
Snapshot Objects from Read/Write Registers Only

This chapter is devoted to *snapshot* objects. Such an object can be seen as a store-collect object whose two operations are atomic. After having defined the concept of a snapshot object, this chapter presents wait-free implementations of it, which are based on atomic read/write registers only. This chapter introduces also the notion of an *immediate snapshot object*, which can be seen as the atomic counterpart of the fast store-collect object presented in the previous chapter.

Keywords Atomic snapshot object · Immediate snapshot object · Infinitely many processes · One-shot object · Read/write system · Recursive algorithm

8.1 Snapshot Objects: Definition

Snapshot object A snapshot object is an object that consists of m components (each component being an atomic read/write register). It provides the processes with two operations denoted update() and snapshot(). The update() operation allows the invoking process to store a new value in a given component, while the snapshot() operation allows it to obtain the values of all the components as if that operation was executed instantaneously.

The invocations of the operations update() and snapshot() are atomic: to an external observer, they appear as if they executed one after the other, each being associated with a point of the time line lying between its start event and its end event (as defined in Chap. 4).

Hence, while store-collect objects are not atomic objects, snapshot objects are. They can consequently be defined by a sequential specification made up of all the traces of update() and snapshot() associated with correct behaviors (each invocation of snapshot() returns the last values of each component that were deposited before that invocation).

Single-writer snapshot object A single-writer snapshot object has one component per object, hence $m = n$, and while a process can read any component it can write

M. Raynal, *Concurrent Programming: Algorithms, Principles, and Foundations*,
DOI: 10.1007/978-3-642-32027-9_8, © Springer-Verlag Berlin Heidelberg 2013

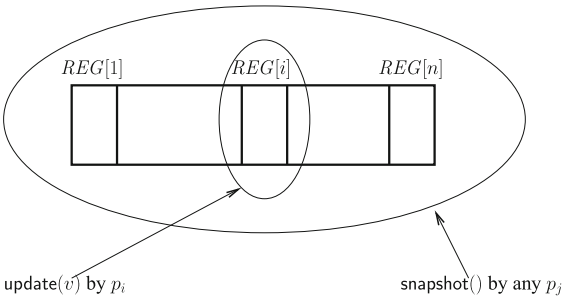

Fig. 8.1 Single-writer snapshot object for n processes

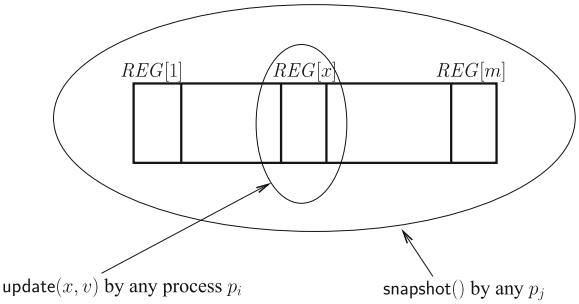

Fig. 8.2 Multi-writer snapshot object with m components

only the component associated with it. In this case, the base atomic read /write registers from which the snapshot object is built are SWMR registers.

As shown in Fig. 8.1, a process p_i invokes update(v) to assign a new value to the SRMW register $REG[i]$ and any process p_j invokes snapshot() to read atomically the whole array $REG[1..n]$.

Multi-writer snapshot object A multi-writer snapshot object is a snapshot object for which each component can be written by any process. It follows that the base atomic read /write registers from which a multi-writer snapshot object is built are MWMR registers.

A multi-writer snapshot object with m components is described in Fig. 8.2. A process p_i invokes update(x, v) to assign the value v to the MWMR component $REG[x]$ and invokes snapshot() to read atomically the whole array $REG[1..m]$.

8.2 Single-Writer Snapshot Object

Assuming a system of n processes, this section presents a wait-free implementation of a single-writer snapshot object due to Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit (1993). This implementation is presented in an incremental way.

8.2.1 An Obstruction-Free Implementation

Let $REG[1..n]$ be the array which is the internal representation of the snapshot object. A simple principle to be able to distinguish different updates of $REG[i]$ by process p_i consists in considering that each atomic register $REG[i]$ is made up of two fields, $REG[i].val$, which contains the last value written by p_i, and $REG[i].sn$, its associated sequence number.

The corresponding algorithm for the update() operation is described in Fig. 8.3. The local variable denoted sn_i, which is initialized to 0, allows p_i to generate sequence numbers.

The algorithm implementing the operation snapshot() is based on what is called a "sequential double collect" which is made up of two consecutive invocations of the function collect(). As we have seen in the previous section, this operation reads asynchronously all the registers of the array $REG[1..n]$ (lines 10–11). (Its trivial implementation considered in the implementations that follow can be replaced by a more efficient adaptive implementation as seen previously).

An invocation of snapshot() repeatedly reads twice the array $REG[1..n]$ (lines 4 and 6 or lines 8 and 6) until two consecutive collects obtain the same sequence number values for each register $REG[j]$ (the local arrays aa_i and bb_i are used to save the values read from the array in the first and second collect, respectively). When, $\forall j : aa_i[j] = bb_i[j]$, the corresponding double collect is said to be *successful* and the algorithm then returns the array of values $[aa_i[1].val, \ldots, aa_i[n].val]$ (line 7).

```
operation update(v) is
(1)   sn_i ← sn_i + 1;
(2)   REG[i] ← ⟨v, sn_i⟩;
(3)   return()
end operation.

operation snapshot() is
(4)   aa_i ← collect();
(5)   repeat forever
(6)      bb_i ← collect();
(7)      if (∀j : aa_i[j] = bb_i[j]) then return(aa_i[1..n].val) end if;
(8)      aa_i ← bb_i
(9)   end repeat
end operation.

internal operation collect() is
(10)  for each j ∈ {1, ..., n} do reg[j] ← REG[j] end for;
(11)  return(reg)
end operation.
```

Fig. 8.3 An obstruction-free implementation of a snapshot object (code for p_i)

Theorem 33 *The algorithms described in Fig. 8.3 define an obstruction-free implementation of an atomic snapshot object.*

Proof Let us first observe that an invocation of an update() operation issued by a correct process always terminate. Let us assume that, after some time, only processes that have invoked snapshot() issue computation steps. It follows that the sequence numbers are no longer modified and consequently any process that executes steps of the operation snapshot() eventually executes a successful double collect and terminates its invocation. It follows that the implementation is obstruction-free.

Let us now show that the implementation provides an atomic snapshot object. To that end, let us define the linearization points on the invocations of the update() and snapshot() operations as follows. As the base registers are atomic, we can consider that their read and write operation occur instantaneously at some point in time.

- Let the linearization point of an update() operation issued by p_j be the time instant when p_j writes $REG[j]$.
- Considering the invocation of a snapshot() operation that returns at line 7, let $\mathsf{collect}_1$ and $\mathsf{collect}_2$ be its last two invocations of collect(). Moreover, let τ_b^i (or τ_e^i) be the time at which $\mathsf{collect}_1$ (or $\mathsf{collect}_2$) read $REG[i]$. As both $\mathsf{collect}_1$ and $\mathsf{collect}_2$ obtain the same sequence number from $REG[i]$, we can conclude that, for any i, no update() operation issued by p_i has terminated and modified $REG[i]$ between τ_b^i and τ_e^i. As $\mathsf{collect}_2$ starts after $\mathsf{collect}_1$ has terminated, it follows that, between $\max(\{\tau_b^i\}_{1\leq i\leq n})$ and $\min(\{\tau_e^i\}_{1\leq i\leq n})$, no atomic register $REG[i]$, $1 \leq i \leq n$, was modified. It is consequently possible to associate with the corresponding invocation of the snapshot() operation a linearization point τ of the time line such that $\max(\{\tau_b^i\}_{1\leq i\leq n}) < \tau < \min(\tau_e^i\}_{1\leq i\leq n})$. Hence, from an external observer point of view, we can consider that the invocation of snapshot() occurred instantaneously at time τ (see Fig. 8.4) after all the invocations of the first update() operation and before the invocations of the second update() operation.

It follows from the previous definition of the linearization points that the operations that terminate define an atomic snapshot object. □

Remark As just noticed, an invocation of update() by a correct process always terminates. Differently, it is possible that invocations of the snapshot() operation never terminate. Let snap be such a non-terminating invocation. This can occur when one or several processes invoke continuouslyupdate() operations in such a way that the termination predicate $\forall j : aa_i[j].sn = bb_i[j].sn$ is never satisfied when checked by the snap. It is important to see that this is not due to the fact that processes crash, but to the fact that processes invoke continuously the update() operation. This is a typical starvation situation that has to be prevented in order to obtain a wait-free construction.

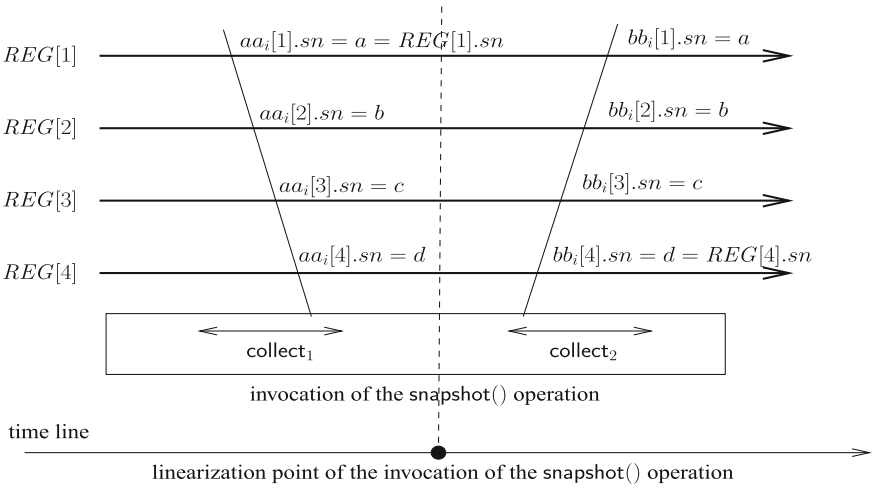

Fig. 8.4 Linearization point of an invocation of the snapshot() operation (case 1)

8.2.2 From Obstruction-Freedom to Bounded Wait-Freedom

A basic principle to obtain a wait-free implementation (when possible) consists in using a helping mechanism (similarly to the one used in Sect. 7.1.3. Here, this "helping" principle can be translated as follows. If the are continuous invocations of the operation update() that could prevent invocations of the snapshot() operation from terminating, these invocations of update() have to help the invocations of snapshot() to terminate. The solution we present relies on this nice and simple idea.

The fact that a double collect is unsuccessful (i.e., does not allow an invocation of the snapshot() operation to terminate) can be attributed to invocations of the update() operation which increase sequence numbers, and consequently make false the test that controls the termination of the algorithm implementing the snapshot() operation.

Let us observe that, if an invocation of snapshot() (say snap) issued by a process p_i sees two distinct invocations of update() issued by the same process p_j (these updates upd_1 and upd_2 write distinct sequence numbers in $REG[j]$), we can conclude that upd_2 was entirely executed during the invocation snap (see Fig. 8.5). This is because the update by p_j of the base atomic register $REG[j]$ is the last operation executed in an update() operation.As the second update upd_2 is entirely executed during the execution of snap (it starts after it and terminates before it), the previous observations suggest requiring upd_2 to help snap. This help can be realized as follows:

- Each update is required to include an "internal" invocation of the snapshot() operation (see Fig. 8.5, where the rectangle inside the invocation upd_2 represents an internal invocation of the snapshot() operation –denoted int_snap– invoked by that update). Let $help$ be the array of values obtained by the invocation int_snap.

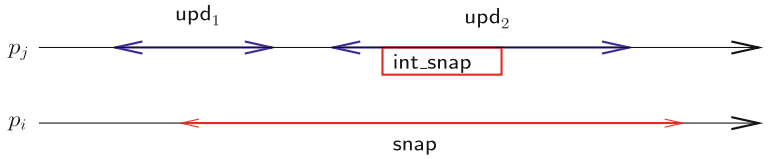

Fig. 8.5 The update() operation includes an invocation of the snapshot() operation

- As the invocation int_snap is entirely overlapped by the invocation snap, snap can borrow the array *help* and return it as its own result. (Notice it is possible that the internal snapshot invocation inside upd_1 may be also entirely overlapped by snap, but there is no way to know this.) This overlapping is important to satisfy the atomicity property; namely, the values returned have to be consistent with the real-time occurrence order of the operation invocations.

The proof will show that such an addition of an invocation of snapshot() to the algorithm implementing the update() operation is not at the price of the wait-free property.

The algorithms implementing the update() and snapshot() algorithms are described in Fig. 8.6. The function collect() is the same as before. A SWMR atomic register $REG[i]$ is now made up of three fields: $REG[i].val$ and $REG[i].sn$ as before, plus the new field $REG[i].help\_array$, whose aim is to contain a helping array as previously discussed. The final version of both update() and snapshot() algorithms is a straightforward extension of their previous attempt versions.

The main novelty lies in the local variable can_help_i that is used by a process p_i when it executes the operation snapshot(). The aim of this set, initialized to $\emptyset$, is to contain the identity of the processes that have terminated an invocation of update() since p_i started its current invocation of the snapshot() operation. Its use is described in the **if** statement at lines 12–17. More precisely, when a double collect is unsuccessful, p_i does the following with respect to each process p_j that made the double collect unsuccessful (i.e., such that $(aa_i[j].sn \neq bb_i[j].sn)$):

- If $j \notin can\_help_i$ (line 15). In this case, p_i discovers that p_j has terminated an invocation of update() since it started its invocation of the snapshot() operation. Consequently, if p_j terminates a new invocation of update() while p_i has not yet terminated its invocation of snapshot(), p_j can help p_i terminate.
- If $j \in can\_help_i$ (line 14). In this case, p_j has entirely executed an update() operation while p_i is executing its snapshot() operation. As we have seen, p_i can benefit from the help provided by p_i by returning the array that p_j stored in $REG[j]$ at the end of its invocation of the operation update().

Theorem 34 *The algorithms described in Fig. 8.6 define a bounded wait-free implementation of an atomic snapshot object.*

Proof Let us first show that the implementation is bounded wait-free. As any invocation of update() contains an invocation of snapshot(), we have to show that any

```
operation update(v) is
(1)  help_array_i ← snapshot();
(2)  sn_i ← sn_i + 1;
(3)  REG[i] ← ⟨v, sn_i, help_array_i⟩;
(4)  return()
end operation.

operation snapshot() is
(5)  can_help_i ← ∅;
(6)  aa_i ← collect();
(7)  repeat forever
(8)      bb_i ← collect();
(9)      if (∀j ∈ {1, ..., n} : aa_i[j].sn = bb_i[j].sn)
(10)       then return(aa_i[1..n].val)
(11)       else for each j ∈ {1, ..., n} do
(12)                if (aa_i[j].sn ≠ bb_i[j].sn) then
(13)                  if (j∈ can_help_i)
(14)                      then return(bb_i[j].help_array)
(15)                      else can_help_i ← can_help_i ∪ {j}
(16)                  end if
(17)                end if
(18)            end for
(19)     end if;
(20) aa_i ← bb_i
(21) end repeat
end operation.

internal operation collect() is
(22) for each j ∈ {1, ..., n} do reg[j] ← REG[j] end for;
(23) return(reg)
end operation.
```

Fig. 8.6 Bounded wait-free implementation of a snapshot object (code for p_i)

invocation of the snapshot() operation issued by a correct process terminates after a bounded number of steps (accesses to atomic registers).

Let us consider that p_i has not returned after having executed he **repeat** loop (lines 7n–21) n times. This means that, each time it has executed that loop, p_i found an identity j such that $aa_i[j].sn \neq bb_i[j].sn$ (line 9), which means that the corresponding process p_j issued a new invocation of the update() operation between the last two collect() issued by of p_i. Each time this occurs, the corresponding process identity j is a new identity added to the set can_help_i at line 15. (If j was already present in can_help_i, p_i would have executed line 14 instead of line 15, and would have consequently terminated its invocation of the snapshot() operation).

As by assumption p_i executes the loop n times, it follows that we have $can\_help_i = \{1, 2, \ldots, n\} \setminus \{i\}$; i.e., this set contains the identities of other processes. It follows that, when it executes the loop for nth time and the test at line 9 is false, whatever

the processes p_j such that $aa_i[j].sn \neq bb_i[j].sn$ at line 12, we necessarily have $j \in can\_help_i$ at line 13, from which it follows that p_i returns at line 14. The implementation is consequently wait-free, as p_i terminates after a finite number of operations on base registers have been executed.

Let us now replace "finite" by "bounded", i.e., let us determine a bound on the number of accesses to base registers. A collect() costs $O(n)$ accesses to base registers. The cost of each iteration of the **for** loop (lines 11–18) is $O(1)$, and there are at most n iteration steps, which means that the cost of that loop is upper-bounded by $O(n)$. Finally, as the enclosing **repeat** loop is executed at most n times, it follows that a process issues at most $O(n^2)$ accesses to base registers when it executes a snapshot() or an update() operation.

Let us now show that the object that is built is atomic. To that end we have to show that, for each execution (and according to the notation introduced in Chap. 4), there is a total order on the invocations of $\widehat{S}$ and update() and snapshot() such that: (1) $\widehat{S}$ includes all the invocations issued by the processes, except possibly, for each process, its the last invocation if that process crashes, (2) $\widehat{S}$ respects the real-time occurrence order on these invocations (i.e., if the invocation op_1 terminates before the invocation op_2 starts, op_1 has to appear before op_2 in $\widehat{S}$), and (3) $\widehat{S}$ respects the semantics of each operation (i.e., a snapshot() invocation has to return, for each process p_j, the value v_j such that, in $\widehat{S}$, there is no upadte() invocation by p_j between update(v_j) and that snapshot() invocation).

The definition of the sequence $\widehat{S}$ relies on (a) the atomicity of the base registers and (b) the fact that the operation snapshot() invokes *sequentially* the underlying function collect(). Let us remember that item (a) means that the read and write operations on the base registers can be considered as being executed instantaneously, each one at a point of the time line, and no two of them at the same time.

The sequence $\widehat{S}$ is built as follows. The linearization point of an invocation of the operation update() is the time at which it atomically executes the write in the corresponding SWMR register (line 3).

The definition of the linearization point of an invocation of the operation snapshot() depends on the line at which it returns:

- The linearization point of an invocation of snapshot() that terminates at line 10 (successful double collect) is at any time time between the end of the first and the beginning of the second of these collect invocations (see Theorem 33 and Fig. 8.4).
- The linearization point of an invocation of snapshot() that terminates at line 14 (i.e., p_i terminates with the help of another process p_j) is defined inductively as follows. (See Fig. 8.7, where a rectangle below an update() invocation represents the internal invocation of snapshot(). The dotted $help\_array$ arrow shows the way an array is conveyed from a successful double collect by a process p_k until an invocation of snapshot() issued by a process p_i.)

  The array (say $help\_array$) returned by p_i was provided by an invocation of update() executed by some process p_j. As already seen, this update() was entirely executed within the time interval of p_i's current invocation of snapshot(). This

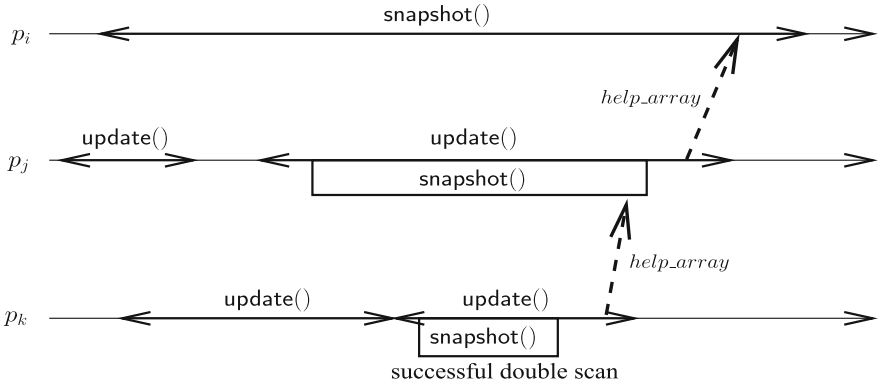

Fig. 8.7 Linearization point of an invocation of the snapshot() operation (case 2)

array was obtained by p_j from a successful double collect, or from another process p_k. If it was obtained from a process p_k, let us consider the way $help\_array$ was obtained by p_k. As there are at most n concurrent invocations of snapshot(), it follows by induction that there is a process p_x that has invoked the snapshot() operation and has obtained $help\_array$ from a successful double collect. Moreover, that invocation of snapshot() was inside an invocation of an update() operation that was entirely executed within the time interval of p_i's current invocation of snapshot().

The linearization point of the invocation of snapshot() issued by p_i is defined from the internal invocation of snapshot() whose successful double collect determined $help\_array$. If several invocations of snapshot() are about to be linearized at the same time, they are ordered according to the total order in which they were invoked.

It follows directly from the previous definition of the linearization points associated with the invocations of the update() and snapshot() operation issued by the processes that $\widehat{S}$ satisfies items (1) and (2) stated at the beginning of the proof. The satisfaction of item (3) comes from the fact that the array returned by any invocation of snapshot() has always been obtained from a successful double collect. □

8.2.3 One-Shot Single-Writer Snapshot Object: Containment Property

A one-shot single-writer snapshot object is a single-writer snapshot object such that each process invokes each operation at most once and always invokes update() before snapshot().

As far as notations are concerned, let v_x be the value written by p_x and $snap_x$ be the array that it later obtained from its invocation of snapshot(). Let us remember that, if a process p_y has not yet invoked update() when $snap_x$ is returned, we

have $snap_x[y] = \perp$. Finally let $(snap_x \preceq snap_y) \equiv \forall i : \big((snap_x[i] \neq \perp) \Rightarrow (snap_x[i] = snap_y[j] = v_j)\big)$. It follows from its atomicity property (linearization points) that a one-shot single-writer snapshot object satisfies the following properties:

- Self-inclusion. For any process p_x: $snap_x[x] = v_x$.
- Containment. For any two processes p_x and p_y: $(snap_x \preceq snap_y) \vee (snap_y \preceq snap_y)$.

Said differently, a one-shot single-writer snapshot object guarantees that all arrays which are returned are totally ordered by the $\preceq$ relation. This containment property is particularly useful and motivated the definition of the one-shot snapshot object.

8.3 Single-Writer Snapshot Object with Infinitely Many Processes

The following implementation is due to M.K. Aguilera (2004).

Computation model The computation model considered in this section is the *finite concurrency model* introduced in Sect. 7.1.2. There are infinitely many processes, but in each time interval only finitely many processes execute operations. Each process p_i has an identity i (an integer), and it is common knowledge that no two distinct process have the same identity. Moreover, the processes have not necessarily consecutive identities.

Adapting the internal representation: the array *REG* To adapt the previous snapshot construction to the finite concurrency model, we first consider that the array *REG* of SWMR atomic registers is a potentially infinite array, starting at *REG*[1], so that an entry *REG*[i] is associated with each "potential" process p_i. For each register *REG*[i], the fields *REG*[i].*sn* and *REG*[i].*val* are initialized to 0 and $\perp$, respectively, where $\perp$ is a default value that no process can write.

Adapting the algorithm for the operation update() Now only the entries *REG*[i] that do correspond to a process p_i that has invoked at least once the operation update() are meaningful. This means that, when a process p_i invokes update(), it has first to indicate in one way or another that, from now on, the entry *REG*[i] is meaningful.

A simple way to do this consists in using a weak counter, denoted *WEAK_CT* as defined in Sect. 7.1.2. This idea is the following: *WEAK_CT* records the highest identity of a process that has invoked the operation update(). In that way, we know that the meaningful entries of the array *REG* are a subset of the entries *REG*[j] such that $1 \leq j \leq WEAK\_CT$. More precisely, these entries correspond to the following set of (process identity, value) pairs:

$$\{(j, REG[j]) \mid (1 \leq j \leq WEAK\_CT) \wedge REG[j].val \neq \perp)\}.$$

The algorithm implementing the update() operation we obtain is described in Fig. 8.8. The lines with the same number in this figure and in the base algorithms

```
operation update(v) is
(N0)   while (WEAK_CT.get_count() < i) do WEAK_CT.increment() end while;
(1)    help_array_i ← snapshot();
(2)    sn_i ← sn_i + 1;
(3)    REG[i] ← ⟨v, sn_i, help_array_i⟩;
(4)    return()
end operation.

operation snapshot() is
(5)    can_help_i ← ∅;
(M.6)  n_init ← WEAK_CT.get_count(); aa_i ← collect(n_init);
(7)    repeat forever
(M.8)     n ← WEAK_CT.get_count(); bb_i ← collect(n);
(9)       if (∀j ∈ {1, ..., n} : aa_i[j].sn = bb_i[j].sn)
(10)        then return(aa_i[1..n].val)
(11)        else for each j ∈ {1, ..., n} do
(12)                if (aa_i[j].sn ≠ bb_i[j].sn) then
(M.13)                if ((j ∈ can_help_i) ∨ (j > n_init))
(14)                   then return(bb_i[j].help_array)
(15)                   else can_help_i ← can_help_i ∪ {j}
(16)                end if
(17)             end if
(18)          end for
(19)    end if;
(20)    aa_i ← bb_i
(21)    end repeat
end operation.

internal operation collect(x) is
(M.22) for each j ∈ {1, ..., x} do reg[j] ← REG[j] end for;
(23)   return(reg)
end operation.
```

Fig. 8.8 Single-writer atomic snapshot for infinitely many processes (code for p_i)

of Fig. 8.6 are the same. As far as the operation update() is concerned, the only difference with respect to the base version is the addition of the first line marked N0.

Adapting the algorithm for the operation snapshot() A first problem that has to be solved consists in making the collect() function always terminate. A simple solution consists in adding an input parameter x to that function, indicating that the collect has to be only from *REG*[1] until *REG*[x]. The value of this parameter is defined as the current value of the counter *WEAK_CT*. The corresponding adaptation of the algorithm implementing the snapshot() operation appears in the lines prefixed by "M" (for modified); more precisely, the lines M.6, M.8, M.13, and M.22 in Fig. 8.8.

A second problem arises when new processes with higher identities invoke update(), causing the counter *WEAK_CT* to increase forever. It is consequently possible that, while it executes the **repeat** loop, an invocation of snapshot() never

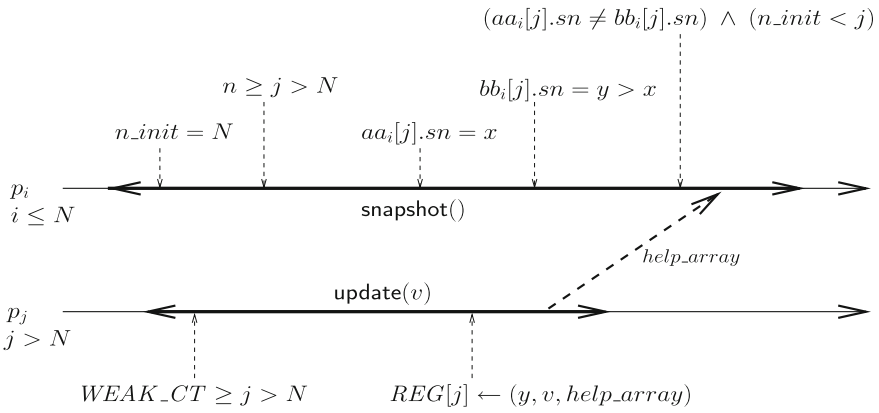

Fig. 8.9 An array transmitted from an update() to a snapshot() operation

finds a process p_j that has terminated two invocations of update() during its invocation of snapshot(): permanently, there are new invocations of update(), but those are issued by new processes with higher and increasing identities.

To solve this problem, let us observe that, if $WEAK\_CT$ increases due to a process p_j, then p_j has necessarily increased it (at line M0 when it executed the update() operation) after p_i started its snapshot operation. So, if n_init is the value of $WEAK\_CT$ when p_i starts invoking snapshot() (see line M.6), this means that we have $j > n\_init$. The solution to the problem (see Fig. 8.9) consists then in replacing the test $j \in could\_help_i$ by the test $j \in could\_help_i \vee j > n\_init$ (line M.3): even if p_j has not executed two update(), $REG[j].help\_array$ can be returned as it was determined after p_i started its invocation of the snapshot() operation.

Remarks As it is written, the returned value (at line 10 or 14) is an array that can contain lots of $\perp$. This depends on the identity of the processes that have previously invoked the update() operation. It is possible to return instead a set of (process identity, value) pairs. On another side, the array can be replaced by a list.

The proof that this is a wait-free implementation of an atomic snapshot object in the finite concurrency model is left as an exercise. The reader can easily remark that the construction is not bounded wait-free (this is because it is not possible a priori to state a bound on the number of iterations of the **while** loop).

8.4 Multi-Writer Snapshot Object

This section presents a multi-writer snapshot algorithm due to D. Imbs and M. Raynal (2011). This implementation is based on a helping mechanism similar to the one used in the previous section. The snapshot object has m components.

8.4.1 The Strong Freshness Property

When we look at the implementation of the operation update(v) described in Fig. 8.6, it appears that the values of the helping array saved by a process p_i in $REG[i].help\_array$ have been written before the write of v into $REG[i]$ (lines 1 and 3 of Fig. 8.6). It follows that, if this array of values is returned at line 13 of Fig. 8.6 by a process p_j, the value $help\_array[i]$ obtained by p_j is older than the value v.

We consider here the following additional property for a multi-writer snapshot object:

- Strong freshness. An invocation of snapshot() which is helped by an operation update(x, v) returns a value for the component x that is at least as recent as v.

The aim of this property is to provide every invocation of the operation snapshot with an array of values that are "as fresh as possible". As we will see, this property will be obtained by separating, inside the operation update(x, v), the write v into $REG[i]$ from the write of the helping array. The corresponding strategy is called "write first, help later". (On the contrary, the implementation described in Fig. 8.6 is based on the strategy of computing a helping array first and later writing atomically both the value v and the helping array).

8.4.2 An Implementation of a Multi-Writer Snapshot Object

Internal representation of the multi-writer snapshot object The internal representation is made up of two arrays of atomic registers. Let us recall that m is the number of components of the snapshot object while n is the number of processes:

- The first array, denoted $REG[1..m]$, is made up of MWMR atomic registers. The register $REG[x]$ is associated with component x. It has three fields $\langle val, pid, sn\rangle$ whose meaning is the following: $REG[x].val$ contains the current value of the component x, while $REG[x].(pid, sn)$ is the "identity" of v. $REG[x].pid$ is the index of the process that issued the corresponding update(x, v) operation, while $REG[x].sn$ is the sequence number associated with this update when considering all updates issued by p_{pid}.
- The second array, denoted $HELPSNAP[1..n]$, is made up of one SWMR atomic register per process. $HELPSNAP[i]$ is written only by p_i and contains a snapshot value of $REG[1..m]$ computed by p_i during its last update() invocation. This snapshot value is destined to help processes that issued snapshot() invocations concurrent with p_i's update. More precisely, if during its invocation of snapshot() a process p_j discovers that it can be helped by p_i, it returns the value currently kept in $HELPSNAP[i]$ as output of its own invocation of snapshot().

The algorithm implementing the operation update(x, v) The algorithm implementing this operation is described at lines 1–4 of Fig. 8.10. It is fairly simple. Let

```
operation update(x, v) is
(1)  sn_i ← sn_i + 1;
(2)  REG[x] ← ⟨v, i, sn_i⟩;
(3)  HELPSNAP[i] ← snapshot();
(4)  return()
end operation.

operation snapshot() is
(5)  can_help_i ← ∅;
(6)  for each x ∈ {1, ··· , m} do aa[x] ← REG[x] end for;
(7)  repeat forever
(8)    for each x ∈ {1, ··· , m} do bb[x] ← REG[x] end for;
(9)    if (∀x ∈ {1, ··· , m} : aa[x] = bb[x]) then return(bb[1..m].val) end if;
(10)   for each x ∈ {1, ··· , m} such that bb[x] ≠ aa[x] do
(11)     let ⟨−, w, −⟩ = bb[x];
(12)     if (w ∈ can_help_i) then return(HELPSNAP[w])
(13)                         else can_help_i ← can_help_i ∪ {w}
(14)     end if
(15)   end for;
(16)   aa ← bb
(17) end repeat
end operation.
```

Fig. 8.10 Wait-free implementation of a multi-writer snapshot object (code for p_i)

p_i be the invoking process. First, p_i increases the local sequence number generator sn_i (initialized to 0) and atomically writes the triple $\langle v, i, sn_i \rangle$ into $REG[x]$. It then computes a snapshot value and writes it into $HELPSNAP[i]$ (line 3).

This constitutes the "write first, help later" strategy. The write of the value v into the component x is executed before the computation and the write of a helping array. The way $HELPSNAP[i]$ can be used by other processes was described previously. Finally, p_i returns from its invocation of update().

It is important to notice that, differently from what is done in Fig. 8.6, the write of v into $REG[x]$ and the write of a snapshot value into $HELPSNAP[i]$ are distinct atomic writes (which access different atomic registers).

The algorithm implementing the operation snapshot(): try first to terminate without help from a successful double collect This algorithm is described at lines 5–17 of Fig. 8.10.

The pair of lines 6 and 8 and the pair of lines 16 and 8 constitute "double collects". Similarly to what is done in Fig. 8.6, a process p_i first issues a double collect to try to compute a snapshot value by itself. The values obtained from the first collect are saved in the local array aa, while the values obtained from the second collect are saved in the local array bb. If $aa[x] = bb[x]$ for each component x, p_i has executed a successful double collect: $REG[1..m]$ contained the same values at any time during the period starting at the end of the first collect and finishing at the beginning of

the second collect. Consequently, p_i returns the array of values $bb[1..m].val$ as the result of its snapshot invocation (line 9).

The algorithm implementing the operation snapshot()**: otherwise, try to benefit from the help of other processes** If the predicate $\forall x : aa[x] = bb[x]$ is false, p_i looks for all entries x that have been modified during its previous double collect. Those are the entries x such that $aa[x] \neq bb[x]$. Let x be such an entry. As witnessed by $bb[x] = \langle -, w, - \rangle$, the component x has been modified by process p_w (line 11).

The predicate $w \in can\_help_i$ (line 12) is the helping predicate. It means that process p_w issued two updates that are concurrent with p_i's current snapshot invocation. As we have seen in the algorithm implementing the operation update(x, v) (line 3; see also Fig. 8.11), this means that p_w has issued an invocation of snapshot() as part of an invocation of update() concurrent with p_i's snapshot invocation. If this predicate is true, the corresponding snapshot value (which has been saved in $HELPSNAP[w]$) can be returned by p_i as output of its snapshot invocation (line 12).

If the predicate is false, process p_i adds the identity w to the set can_help_i (line 13). Hence, can_help_i (which is initialized to $\emptyset$, line 1) contains identities y indicating that process p_y has issued its last update while p_i is executing its snapshot operation. Process p_i then moves the array bb into the array aa (line 16) and re-enters the **repeat**. (As already indicated, the lines 16 and 08 constitute a new double scan.)

On the "write first, help later" strategy As we can see, this strategy is very simple. It has several noteworthy advantages:

- This strategy first allows atomic write operations (at line 2 and line 3) to write values into base atomic registers $REG[r]$ and $HELPSNAP[i]$ that have a smaller size than the values written in the single-writer snapshot object implementation of Fig. 8.6 (where an atomic write into $REG[x]$ is on a triple of values). Atomic writes of smaller values allow for more efficient solutions.
- Second, this simple strategy allows the atomic writes into the base atomic registers $REG[x]$ and $HELPSNAP[i]$ to be not synchronized (while they are strongly synchronized in the single-writer snapshot implementation of Fig. 8.6, where they are pieced into a single atomic write).

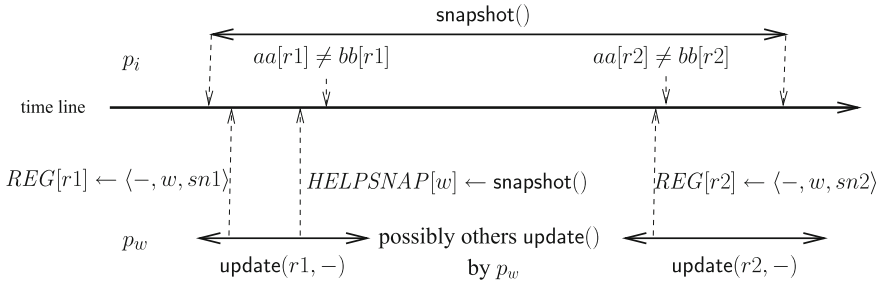

Fig. 8.11 A snapshot() with two concurrent update() by the same process

- Finally, as shown in the proof, the "write first, help later" strategy allows the invocations of snapshot() to satisfy the strong freshness property (i.e., to return component values that are "as fresh as possible").

Cost of the implementation This section analyses the cost of the operations update() and snapshot() in terms of the number of base atomic registers that are accessed by a read or write operation.

- Operation snapshot().
  - Best case. In the best case an invocation of the operation snapshot() returns after having read only twice the array $REG[1..m]$. The cost is then $2m$.
  - Worst case. Let p_i be the process that invoked operation snapshot(). The worst case is when a process returns at line 12 and the local array can_help_i contains $n-1$ identities: an identity from every process but p_i. In that case, p_i has read $n+1$ times the array $REG[1..m]$ and, consequently, has accessed $(n+1)m$ times the shared memory.
- The cost of an update operation is the cost of a snapshot operation plus 1.

It follows that the cost of an operation is $O(n \times m)$.

8.4.3 Proof of the Implementation

Definition 1 The array of values $[v_1, \ldots, v_m]$ returned by an invocation of snapshot() is *well defined* if, for each x, $1 \leq x \leq m$, the value v_x has been read from $REG[x]$.

Definition 2 The values returned by an invocation of snapshot() are *mutually consistent* if there is a time at which they were simultaneously present in the snapshot object.

Definition 3 The values returned by an invocation of snapshot() are strongly fresh if, for each x, $1 \leq x \leq m$, the value v_x returned for component x is not older than the last value written into $REG[x]$ before the snapshot invocation. (Let us recall that, as each $REG[x]$ is an atomic register, its read and write operations can be totally ordered in a consistent way. The term "last" is used with respect to this total order).

Definition 4 Let snap be an invocation of snapshot() issued by a process p_i.

- The invocation snap is 0-helped if it terminates with a successful double collect (line 9 of Fig. 8.10).
- The invocation snap is 1-helped if it terminates by returning $HELPSNAP[w1]$ (line 12 of Fig. 8.10) and the values in $HELPSNAP[w1]$ come from a successful double collect by p_{w1} (i.e., the values in $HELPSNAP[w1]$ have been computed at line 3 by the invocation of snapshot() inside the invocation of update() issued by p_{w1}).

- The invocation snap is 2-helped if it terminates by returning $HELPSNAP[w1]$ (line 12) and the values in $HELPSNAP[w1]$ come from a 1-helped snapshot() operation invoked by a process p_{w2} at line 03.
- For the next values of h, the h-helped notion is similarly defined by induction.

Lemma 14 *The values returned by a* 0*-helped invocation of* snapshot() *are well defined, mutually consistent, and strongly fresh.*

Proof Let us consider a 0-helped invocation of the operation snapshot(). As it terminates at line 9, the array of values returned are the values in the array $bb[1..m]$. It follows from line 8 that this array is well defined (its values are from the array $REG[1..m]$).

Mutual consistency and strong freshness follow from the fact that the termination of the invocation of snapshot() is due to a successful double collect. More precisely, the values kept in $bb[1..m]$ were present simultaneously in $REG[1..m]$ during the period starting at the end of the first collect and ending at the beginning of the second collect (mutual consistency) and, for any entry x, the value returned from $REG[x]$ is not older than the value kept in $REG[x]$ when the invocation of snapshot() started (strong freshness). □

Lemma 15 *The values returned by a* 1*-helped* snapshot() *operation are well defined, mutually consistent, and strongly fresh.*

Proof Let snap be a 1-helped invocation of snapshot(). It follows from the definition of 1-helped invocation that (a) the array returned by snap (namely the value read by p_i from $HELPSNAP[w1]$) was computed by a snapshot operation snap′ invoked inside an invocation of update() issued by p_{w1}, and (b) snap′ is 0-helped (i.e., $HELPSNAP[w1]$ comes from a successful double collect). It then follows from Lemma 14 that the values in $HELPSNAP[w1]$ are well defined and mutually consistent.

Let us now show that the values contained in $HELPSNAP[w1]$ are strongly fresh; i.e., for each x, the value in $HELPSNAP[w1][x]$ is not older than the last value written into $REG[x]$ before snap started. Let up1 and up2 be the two updates issued by p_{w1} whose sequence numbers are $sn1$ and $sn2$ (let us observe that, due to the test of line 12, these updates do exist). Moreover, snap′ is an internal invocation issued by an invocation up′ of update() from p_{w1} whose sequence number sn is such that $sn1 \leq sn \leq sn2$.

As the value of $HELPSNAP[w1]$ that is returned is computed during up′, and up′ was invoked by p_{w1} after up1, the values obtained from REG and assigned to $HELPSNAP[w1]$ were read after up1 started. We claim that up1 started after snp. It follows from this claim that the values in $HELPSNAP[w1]$ that are returned were read from REG after snap started, which proves the lemma.

Proof of the claim. Let $r1$ be the component that entailed the addition of the identity $w1$ to can_help_i at line 13 during the execution of snap. Moreover, let $aa1[r1]$ and $bb1[r1]$ be the values in the local arrays aa and bb that entailed this addition. Hence, we have $aa1[r1] \neq bb1[r1]$ (line 10) and $bb1[r1] = \langle -, w1, - \rangle$.

As snap has read $REG[r1]$ twice (first to obtain $aa1[r1]$, and then to obtain $bb1[r1]$) and $aa1[r1] \neq bb1[r1]$, it follows that up1 started after the start of snp, which concludes the proof of the claim. □

Lemma 16 *For any h, the values returned by an h-helped invocation of* snapshot() *are well-defined, mutually consistent, and strongly fresh.*

Proof The proof is by induction. The base case is $h = 0$ (Lemma 14). Assuming that the lemma is satisfied up to $h - 1$, the proof for h is similar to the proof for $h = 1$ that relies on the fact that we have a proof for the case $h - 1 = 0$ (Lemma 15). □

Lemma 17 *Wait-freedom. Any invocation of* update() *or* snapshot() *issued by a correct process terminates.*

Proof Let us first observe that, if every snapshot() operation issued by a correct process terminates, then all its update() operations terminate. Hence, the proof only has to show that all snapshot() operations issued by a correct process terminate.

Let us consider an invocation of snapshot() issued by a correct process p_i. If, when p_i executes line 9, the predicate is true, the invocation terminates. So, we have to show that, if the predicate of line 9 is never satisfied, then the predicate of line 12 eventually becomes true. As the predicate of line 9 is never satisfied, each time p_i executes the loop body, there is a component x such that $aa[x] \neq bb[x]$. The process p_k that modified $REG[k]$ between the two readings by p_i entails the addition of its identity k to can_help_i (where k is extracted from $bb[x]$). In the worst case, $n - 1$ identities (one per process except p_i, because it cannot execute an update operation while it executes a snapshot operation) can be added to can_help_i while the predicate of line 12 remains false. But, once can_help_i contains one identity per process (but p_i), the test of line 12 necessarily becomes satisfied, which proves the lemma. □

The next lemma shows that the implementation is atomic, i.e., that the invocations of update() and snapshot() issued by the processes during a run (except possibly the last operation issued by faulty processes) appear as if they have been executed one after the other, each one being executed at some point of the time line between its start event and its end event.

Lemma 18 *The algorithms described in Fig. 8.10 implement an atomic multi-writer snapshot object.*

Proof The proof consists in associating with each invocation inv of update() and snapshot() a single point of the time line denoted $\ell p(\text{inv})$ (linearization point of inv) such that:

- $\ell p(\text{inv})$ lies between the beginning (start event) of inv and its end (end event),
- no two operations have the same linearization point,
- the sequence of the operation invocations defined by their linearization points is a sequential execution of the snapshot object.

So, the proof consists in (a) an appropriate definition of the linearization points and (b) showing that the associated sequence satisfies the specification of the snapshot object.

- Point (a): definition of the linearization points.

  The linearization point of each operation invocation (except possibly the last operation of faulty processes) is defined as follows:

  – The linearization point of an invocation of update$(r, -)$ is the linearization point of its write of $REG[r]$ (line 2). (Let us remember that, as the underlying atomic registers are atomic, they have well-defined linearization points).

  – The linearization point of an invocation psp of snapshot() depends on the line at which the return() statement is executed:

    * Case 1: psp returns at line 9 due a successful double collect (i.e., psp is 0-helped). Its linearization point is any point of the time line between the first and the second collect of that successful double collect.

    * Case 2: psp returns at line 12 (i.e., psp is h-helped with $h \geq 1$). In this case, the array of values returned by psp was computed by some update operation at line 3. Moreover, whatever the value of h, this array was computed by a successful double collect executed by some process p_z. When considering this successful double collect, ℓp(psp) is placed between the end of its first collect and the beginning of its second collect.

  If two operations are about to be linearized at the same point, they are arbitrarily ordered (e.g., according to the identities of the processes that issued them).
  It follows from the previous linearization point definitions that each invocation of an operation is linearized between its beginning and its end, and no two operations are linearized at the same point.

- Point (b): the sequence of invocations of update() and snapshot() defined by their linearization points satisfies the specification of the snapshot object.
  This follows directly from Lemma 15 which showed that the values returned by every snapshot operation are well defined, mutually consistent, and strongly fresh. □

Theorem 35 *The implementation described in Fig. 8.10 is a bounded wait-free implementation of an atomic multi-writer snapshot object which also satisfies the strong freshness property.*

Proof The proof that the implementation satisfies the consistency property of a snapshot object follows from Lemma 18. Wait-freedom follows from Lemma 17. The freshness property follows from the definition of the linearization points given in Lemma 18. Finally, the fact that the implementation is bounded wait-free follows from the fact that an operation costs at most $O(m \times n)$ accesses to base atomic registers. □

8.5 Immediate Snapshot Objects

The notion of an immediate snapshot object is due to E. Borowsky and E. Gafni (1993).

8.5.1 One-Shot Immediate Snapshot Object: Definition

A one-shot *immediate snapshot* object is a one-shot snapshot object where the update() and snapshot() are fused in a single operation denoted update_snapshot() and such that each process invokes at most once that operation. (A similar fusion of write and read operations has already been done in Sect. 7.3, where the operations store() and collect() were pieced together to give rise to the operation store_collect()).

When a process p_i invokes update_snapshot(v), it deposits v and obtains a view $view_i$ made up of (process identity, value) pairs. From an external observer point of view, everything has to appear as if the operation was executed instantaneously.

More formally, a one-shot immediate snapshot object is defined by the following properties. Let $view_i$ denote the set of (process identity, value) pairs returned by p_i when it invokes update_snapshot(v_i).

- Liveness. An invocation of update_snapshot() by a correct process terminates.
- Self-inclusion. $(i, v_i) \in view_i$.
- Containment. $\forall i, j : view_i \subseteq view_j$ or $view_j \subseteq view_i$.
- Immediacy. $\forall i, j :$ if $(j, v_j) \in view_i$ then $view_j \subseteq view_i$.

The first three properties (liveness, self-inclusion, and containment) are the properties which define a one-shot snapshot object. When considering only these properties, update_snapshot(v_i) could be implemented by the invocation update(v_i) immediately followed by the invocation snapshot().

The immediacy property is not satisfied by this implementation, which shows the fundamental difference between snapshot and immediate snapshot.

8.5.2 One-Shot Immediate Snapshot Versus One-Shot Snapshot

Figure 8.12 depicts three processes p_1, p_2, and p_3. Each process p_i first invokes update(v_i) and later invokes snapshot(). (In the figure, process identities appear as subscripts in the operation invoked).

According to the specification of the one-shot snapshot object, the invocation $\text{snapshot}_1()$ returns the view $\{(1, v_1), (2, v_2)\}$, while both the invocations $\text{snapshot}_2()$ issued by p_2 and $\text{snapshot}_3()$ issued by p_3 return the view $\{(1, v_1), (2, v_2), (3, V_3)\}$. This means that it is possible to associate with this execution the following sequence of operation invocations $\widehat{S}$:

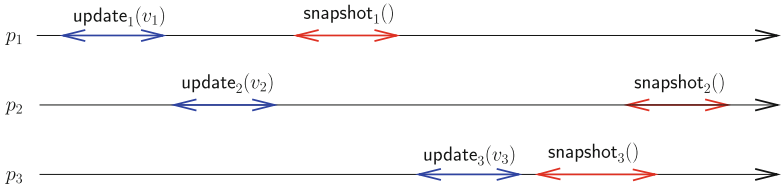

Fig. 8.12 An execution of a one-shot snapshot object

$$\mathsf{update}_1(v_1)\ \ \mathsf{update}_2(v_2)\ \ \mathsf{snapshot}_1()\ \ \mathsf{update}_3(v_3)\ \ \mathsf{snapshot}_2()\ \ \mathsf{snapshot}_3(),$$

which belongs to the sequential specification of a one-shot snapshot object and shows, consequently, that this execution is atomic.

Figure 8.13 shows the same processes where the invocations $\mathsf{update}_i()$ and $\mathsf{snapshot}_i()$ issued by p_i are replaced by a single invocation $\mathsf{update\_snapshot}_i()$ (that starts at the same time as $\mathsf{update}_i()$ starts and terminates at the same time as $\mathsf{snapshot}_i()$).

As $\mathsf{update\_snapshot}_1(1)$ terminates before $\mathsf{update\_snapshot}_3(3)$ starts, we necessarily have $(1, v_1) \in view_3$ and $(3, v_3) \notin view_1$. More generally, each of the five items that follow defines a set of correct outputs for the execution of Fig. 8.13 (said differently, this execution is non-deterministic in the sense that its outputs are defined by any one of these five items):

1. $view_1 = \{(1, v_1)\}$, $view_2 = \{(1, v_1), (2, v_2)\}$, and $view_3 = \{(1, v_1), (2, v_2), (3, v_3)\}$,
2. $view_1 = view_2 = \{(1, v_1), (2, v_2)\}$, and $view_3 = \{(1, v_1), (2, v_2), (3, v_3)\}$,
3. $view_2 = \{(2, v_2)\}$, $view_1 = \{(1, v_1), (2, v_2)\}$, and $view_3 = \{(1, v_1), (2, v_2), (3, v_3)\}$,
4. $view_1 = \{(1, v_1)\}$, and $view_2 = view_3 = \{(1, v_1), (2, v_2), (3, v_3)\}$, and
5. $view_1 = \{(1, v_1)\}$, $view_3 = \{(1, v_1), (3, v_3)\}$, and $view_2 = \{(1, v_1), (2, v_2), (3, v_3)\}$.

When $view_1$, $view_2$ and $view_3$ are all different (item 1), everything appears as if the three invocations of update_snapshot() have been executed sequentially (and

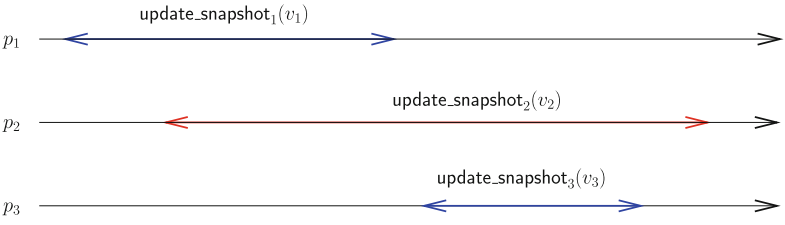

Fig. 8.13 An execution of an immediate one-shot snapshot object

consistently with their real-time occurrence order). When two of them are equal, e.g., $view_1 = view_2 = \{(1, v_1), (2, v_2)\}$ (item 2), everything appears as if the invocations update_snapshot$_1$(1) and update_snapshot$_2$(2) have been issued at the very same time, both before update_snapshot$_3$(). This possibility of simultaneity is the very essence of the "immediate" snapshot abstraction. It also shows that an immediate snapshot object is not an atomic object.

Theorem 36 *A one-shot immediate snapshot object satisfies the following property: if* $(i, -) \in view_j$ *and* $(j, -) \in view_i$, *then* $view_i = view_j$.

Proof If $(j, -) \in view_i$ (theorem assumption), we have $view_j \subseteq view_i$ from the immediacy property. Similarly, $(i, -) \in view_j$ implies that $view_i \subseteq view_j$. It trivially follows that $view_i = view_j$ when $(j, -) \in view_i$ and $(i, -) \in view_j$. □

Set-linearizability The previous theorem states that, while its operations appear as if they were executed instantaneously, an immediate snapshot object is not an atomic object. This is because it is not always possible to totally order all its operations. The immediacy property states that, from a logical time point of view, it is possible that several invocations appear as being executed simultaneously (they then return the same view), making it impossible to consider that one occurred before the other. This means that an immediate snapshot object has no sequential specification. We then say that these invocations are *set-linearized* at the same point of the time line, and the notion of a linearization point is replaced by *set-linearization*.

Hence, differently from a snapshot object, the specification of an immediate snapshot object allows for concurrent operations from an external observer point of view. It then requires that the invocations which are set-linearized at the same point do return the very same view.

8.5.3 An Implementation of One-Shot Immediate Snapshot Objects

This section describes a one-shot immediate snapshot implementation due to E. Borowsky and E. Gafni (1993).

Underlying data structure This implementation is based on two arrays of SWMR atomic registers:

- The array $REG[1..n]$, which is initialized to $[\bot, \ldots, \bot]$, is such that $REG[i]$ is used by p_i to store its value v_i.
- $LEVEL[1..n]$ is initialized to $[n + 1, \ldots, n + 1]$. $LEVEL[i]$ can be written only by p_i. A process p_i uses also a local array $level_i[1..n]$ to keep the last values it (asynchronously) reads from $LEVEL[1..n]$. A register $LEVEL[i]$ contains at most n distinct values (from $n+1$ until 1), which means that it requires $b = \lceil \log_2(n+1) \rceil$ bits.

```
operation update_snapshot(v_i) is
(1)  REG[j] ← v_i;
(2)  repeat LEVEL[i] ← LEVEL[i] − 1;
(3)        for each j ∈ {1, ..., n} do level_i[j] ← LEVEL[j] end for;
(4)        set_i ← {x | level_i[x] ≤ level_i[i]}
(5)  until (|set_i| ≥ level_i[i]) end repeat;
(6)  let view_i = { (x, REG[x]) | x ∈ set_i };
(7)  return(view_i)
end operation.
```

Fig. 8.14 An algorithm for the operation update_snapshot() (code for process p_i)

Underlying principles and the algorithm implementing the operation update_snapshot() Let us consider the image of a stairway made up of $n+1$ stairs. Initially all the processes stand at the highest stair (i.e., the stair whose number is $n+1$). (This is represented in the algorithm by the initial values of the *LEVEL* array; namely, for any process p_j, we have $LEVEL[j] = n+1$).

The algorithm, which is described in Fig. 8.14, is based on the following idea. When a process p_i invokes update_snapshot(), it first deposits its value v_i in $REG[i]$ (line 1). Then it descends along the stairway (lines 2–5), going from step $LEVEL[i]$ to step $LEVEL[i]-1$ until it attains a step k such that there are exactly k processes (including itself) stopped on steps 1 to k. The identities of these k processes are saved in the local set set_i. It then returns the view made up of the k pairs $(x, REG[x])$ such that $x \in set_i$ (the k processes which stopped on one of the steps 1 to k).

To catch the underlying intuition and understand how this idea works, let us consider two extremal cases in which k processes invoke the update_snapshot() operation:

- Sequential case.

  In this case, the k processes invoke the operation sequentially; i.e., the next invocation starts only after the previous one has returned. It is easy to see that the first process p_{i_1} that invokes the update_snapshot() operation proceeds from step $n+1$ until step number 1, and stops at this step. Then, the process p_{i_2} starts and descends from step $n+1$ until step number 2, etc., and the last process p_{i_k} stops at step k.

  Moreover, the set returned by p_{i_1} is $\{i_1\}$, the set returned by p_{i_2} is $\{i_1, i_2\}$, etc., the set returned by p_{i_k} being $\{i_1, i_2, \ldots, i_k\}$. These sets trivially satisfy the inclusion property.

- Synchronous case.

  In this case, the k processes proceed synchronously. They all, simultaneously, descend from step $n+1$ to step n, then from step n to step $n-1$, etc., and they all stop at step number k because there are then k processes at steps from 1 to k (they all are on the same kth step).

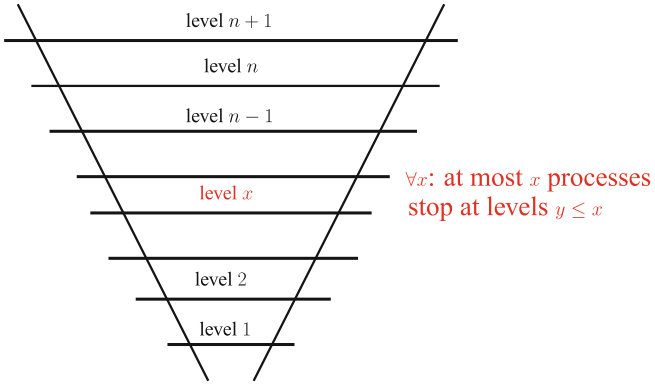

Fig. 8.15 The levels of an immediate snapshot objects

It follows that all the processes return the very same set of participating processes, namely, the set including all of them $\{i_1, i_2, \ldots, i_k\}$, and the inclusion property is trivially satisfied.

Other cases, where the processes proceed asynchronously and some of them crash, can easily be designed. The general case is described in Fig. 8.15. If p_i stops at level x, its view includes all the pairs $\langle j, v_j \rangle$ such that p_j returns or crash at level x.

How levels are implemented The main question is now how to make this idea operational? This is done by three statements (Fig. 8.14). Let us consider a process p_i:

- First, when it is standing at a given step $LEVEL[i]$, p_i reads the steps at which the other processes are (line 3). The aim of this asynchronous reading is to allow p_i to compute an approximate global state of the stairway. Let us notice that, as a process p_j can only go downstairs, $level_i[j]$ is equal or smaller to the step $k = LEVEL[i]$ on which p_j currently is. It follows that, despite the fact the global state obtained by p_i is approximate, set_i can be safely used by p_i.
- Then (line 4), p_i uses the global state of the stairway it has obtained to compute the set (denoted set_i) of processes that, from its point of view, are standing at a step between $LEVEL[1]$ and $LEVEL[i]$ (the step where p_i currently is).
- Finally (line 5), if set_i contains $k = LEVEL[i]$ or more processes, p_i returns the corresponding view (line 6–7). Otherwise, it proceeds to the next stair $LEVEL[i] - 1$ (line 2).

Two preliminary lemmas are proved before the main theorem.

Lemma 19 *Let $set_i = \{x \mid level_i[x] \leq LEVEL[i]\}$ (as computed at line 4). For any process p_i, the predicate $|set_i| \leq LEVEL[i]$ is always satisfied at line 5.*

Proof Let us first observe that $level_i[i]$ and $LEVEL[i]$ are always equal at lines 4 and 5. Moreover, any $LEVEL[j]$ register can only decrease, and for any pair (i, j) we have $LEVEL[j] \leq level_i[j]$.

The proof is by contradiction. Let us assume that there is at least one process p_i such that $|set_i| = |\{x \mid level_i[x] \leq LEVEL[i]\}| > LEVEL[i]$. Let k be the current value of $LEVEL[i]$ when this occurs.$|set_i| > k$ and $LEVEL[i] = k$ mean that at least $k+1$ processes have progressed at least to stair k. Moreover, as any process p_j descends one stair at a time (it proceeds from stair $LEVEL[j]$ to stair $LEVEL[j]-1$ without skipping stairs),at least $k+1$ processes have proceeded from stair $k+1$ to stair k.

Among the at least $k+1$ processes that are on a stair $\leq k$, let p_ℓ be the last process that updated its $LEVEL[\ell]$ register to $k+1$ (due to the atomicity of the base registers, there is such a last process). When p_ℓ was on stair $k+1$ (we then had $LEVEL[\ell] = k+1$), it obtained at line 4 a set set_ℓ such that $|set_\ell| = |\{x \mid level_\ell[x]| \leq LEVEL[\ell]\} \geq k+1$ (this is because at least $k+1$ processes have proceeded to the stair $k+1$, and as p_ℓ is the last of them, it read a value smaller than or equal to $k+1$ from its own $LEVEL[\ell]$ register and the ones of those processes). As $|set_\ell| \geq k+1$, p_ℓ stopped descending the stairway at line 5, at stair $k+1$. It then returned, contradicting the initial assumption stating that it progresses until stair k. □

Lemma 20 *If p_i halts at stair k, we then have $|set_i| = k$. Moreover, set_i is composed of the processes that are at a stair $k' \leq k$.*

Proof Due to Lemma 19, we always have $|set_i| \leq LEVEL[i]$, when p_i executes line 5. If it stops, we also have $|set_i| \geq LEVEL[i]$ (test of line 5). It follows that $|set_i| = LEVEL[i]$. Finally, if k is p_i's current stair, we have $LEVEL[i] = k$ (definition of $LEVEL[i]$ and line 2). Hence, $|set_i| = k$.

The fact that set_i is composed of the identities of the processes that are at a stair smaller than or equal to k follows from the very definition of set_i (namely,$set_i = \{x \mid level_i[x] \leq LEVEL[i]\}$), the fact that,for any x,$level_i[x] \leq LEVEL[x]$, and the fact that a process never climbs the stairway (it either halts on a stair,line 5, or descends to the next one, line 2). □

Theorem 37 *The algorithm described in Fig. 8.14 is a bounded wait-free implementation of a one-shot immediate snapshot object.*

Proof Let us observe that (1) $LEVEL[i]$ is monotonically decreasing, and (2) at any time, set_i is such that $|set_i| \geq 1$ (because it contains at least the identity i). It follows that the **repeat** loop always terminates (in the worst case when $LEVEL[i] = 1$). Hence, the algorithm is wait-free. Moreover, p_i executes the **repeat** loop at most n times, and each computation inside the loop includes n reads of atomic base registers. It follows that $O(n^2)$ is an upper bound on the number of read/write operations on base registers issued in an invocation of update_snapshot(). The algorithm is consequently bounded wait-free.

The self-inclusion property is a direct consequence of the way set_i is computed (line 4): trivially,the set $\{x|level_i[x] \leq level_i[i]\}$ always contains i.

For the containment property, let us consider two processes p_i and p_j that stop at stairs k_i, and k_j, respectively. Without loss of generality, let $k_i \leq k_j$. Due to Lemma 20, there are exactly k_i processes on the stairs 1 to k_i, and k_j processes on stairs 1 to $k_j \leq k_i$. As no process backtracks on the stairway (a process proceeds downwards or stops), the set of k_j processes returned by p_j includes the set of k_1 processes returned by p_i.

Let us finally consider the immediacy property. Let us first observe that a process deposits its value before starting its descent of the stairway (line 1), from which it follows that, if $j \in set_i$, $REG[j]$ contains the value v_j deposited by p_j. Moreover, it follows from lines 4 and 5 that, if a a process p_j stops at a stair k_j and then $i \in set_j$, then p_i stopped at a stair $k_i \leq k_j$. It then follows from Lemma 20 that the set set_j returned by p_j includes the set set_i returned by p_i, from which follows the immediacy property. □

8.5.4 A Recursive Implementation of a One-Shot Immediate Snapshot Object

This section describes a recursive implementation of a one-shot immediate snapshot object due to E. Gafni and S. Rajsbaum (2010). This construction can be seen as a recursive formulation of the previous iterative algorithm.

Underlying data structure As we are about to see, when considering distributed computing, an important point that distinguishes distributed recursion from sequential recursion on data structures lies in the fact that the recursion parameter is usually the number n of processes involved in the computation. The recursion parameter is used by a process to compute a view of the concurrency degree among the participating processes.

The underlying data structure representing the immediate snapshot object consists of a shared array $REG[1..n]$ such that each $REG[x]$ is an array of n SWMR atomic registers. The aim of $REG[x]$, which is initialized to $[\bot, \ldots, \bot]$, is to contain the view obtained by the processes that see exactly x other processes in the system. For any x, $REG[x]$ is accessed only by the processes that attain recursion level x and the atomic register $REG[x][i]$ can be read by all these processes but can be written only by p_i.

The recursive algorithm implementing the operation update_snapshot() The algorithm is described in Fig. 8.16. Its main call is an invocation of rec_update_snapshot(n, v_i), where n is the initial value of the recursion parameter and v_i the value that p_i wants to deposit into the immediate snapshot object (line 1). This call is said to occur at recursion level n. More generally, an invocation rec_update_snapshot$(x, -)$ is said to occur at recursion level x. Hence, the recursion levels are decreasing from level n to level $n-1$, then to level $n-2$, etc. (Actually, a recursion level corresponds to what was called a "level" in Sect. 8.5.3.)

```
operation update_snapshot(v_i) is
(1)  my_view_i ← rec_update_snapshot(n, v_i)
(2)  return(my_view_i)
end operation.

operation rec_update_snapshot(x, v) is
       % x is the recursion parameter (n ≥ x ≥ 1) %
(3)  REG[x][i] ← v;
(4)  for each j ∈ {1, ..., n} do reg_i[j] ← REG[x][j] end for;
(5)  view_i ← { (j, reg_i[j]) | reg_i[j] ≠ ⊥ };
(6)  if (|view_i| = x) then res_i ← view_i
(7)                    else res_i ← rec_update_snapshot(x − 1, v)
(8)  end if;
(9)  return(res_i)
end operation.
```

Fig. 8.16 Recursive construction of a one-shot immediate snapshot object (code for process p_i)

When it invokes rec_update_snapshot(x, v), p_i first writes v into $REG[x][i]$ and reads asynchronously the content of $REG[x][1..n]$ (lines 3–4, let us notice that these lines implement a store-collect). Hence, the array $REG[x][1..n]$ is devoted to the xth recursion level.

Then, p_i computes the view $view_i$ obtained from $REG[x][1..n]$ (line 5). Let us remark that, as the recursion levels are decreasing and there are at most n participating processes, the set $view_i$ contains the values deposited by $n' = |view_i|$ processes, where n' is the number of processes that, from p_i's point of view, have attained recursion level x.

If p_i sees that exactly x processes have attained the recursion level x (i.e., $n' = x$), it returns $view_i$ as the result of its invocation of the immediate snapshot object (lines 6 and 9). Otherwise, fewer than x processes have attained recursion level x and consequently p_i invokes recursively rec_update_snapshot$(x - 1, v)$ (line 7) in order to attain a recursion level $x' < x$ accessed by exactly x' processes. It will stop its recursive invocations when it attains such a recursion level (in the worst case, $x' = 1$).

Theorem 38 *The algorithm described in Fig. 8.16 is a wait-free construction of an immediate snapshot object. Its step complexity (number of shared memory accesses) is $O(n(n - |res| + 1))$, where res is the set returned by* update_snapshot(v).

Proof Claim C. If at most x processes invoke rec_update_snapshot$(x, -)$ then (a) at most $(x - 1)$ processes invoke rec_update_snapshot$(x - 1, -)$ and (b) at least one process stops at line 6 of its invocation rec_update_snapshot$(x, -)$.

Proof of claim C. Assuming that at most x processes invoke update_snapshot $(x, -)$, let p_k be the last process that writes into $REG[x][1..n]$. We necessarily have $|view_k| \leq x$.If p_k finds $|view_k| = x$, it stops at line 6. Otherwise, we have $|view_k| < x$ and p_k invokes rec_update_snapshot$(x - 1, -)$ at line 7. But in that

case, as p_k is the last process that wrote into the array $REG[x][1..n]$, it follows from $|view_k| < x$ that fewer than x processes have written into $REG[x][1..n]$, and consequently, at most $(x-1)$ processes invoke rec_update_snapshot$(x-1, -)$. End of the proof of claim C.

To prove the termination property, let us consider a correct process p_i that invokes update_snapshot(v_i). Hence, it invokes rec_update_snapshot$(n, -)$. It follows from Claim C and the fact that at most n processes invoke rec_update_snapshot $(n, -)$ that either p_i stops at that invocation or belongs to the set of at most $n-1$ processes that invoke rec_update_snapshot$(n-1, -)$. It then follows by induction from the claim that if p_i has not stopped during a previous invocation, it is the only process that invokes rec_update_snapshot(1). It then follows from the text of the algorithm that it stops at that invocation.

The proof of the self-inclusion property is trivial. Before stopping at recursion level x (line 6), a process p_i has written v_i into $REG[x][i]$ (line 3), and consequently we have then $(i, v_i) \in view_i$, which concludes the proof of the self-inclusion property.

To prove the self-containment and immediacy properties, let us first consider the case of two processes that return at the same recursion level x. If a process p_i returns at line 6 of recursion level x, let $view_i[x]$ denote the corresponding value of $view_i$. Among the processes that stop at recursion level x, let p_i be the last process which writes into $REG[x][1..n]$. As p_i stops, this means that $REG[x][1..n]$ has exactly x entries different from $\perp$ and (due to Claim C) no more of its entries will be set to a non-$\perp$ value. It follows that, as any other process p_j that stops at recursion level x reads x non-$\perp$ entries from $REG[x][1..n]$, we have $view_i[x] = view_j[x]$ which proves the properties.

Let us now consider the case of two processes p_i and p_j that return at line 6 of recursion level x and y, respectively, with $x > y$; i.e., p_i returns $view_i[x]$ while p_j returns $view_j[y]$. The self-containment follows then from $x > y$ and the fact that p_j has written into all the arrays $REG[z][1..n]$ with $n \geq z \geq y$, from which we conclude that $view_j[y] \subseteq view_i[x]$. Moreover, as $x > y$, p_i has not written into $REG[y][1..n]$ while p_j has written into $REG[x][1..n]$, and consequently $(j, v_j) \in view_i[x]$ while $(i, v_i) \notin view_j[y]$, from which the containment and immediacy properties follow.

As far as the number of shared memory accesses is concerned we have the following. Let *res* be the set returned by an invocation of rec_update_snapshot$(n, -)$. Each recursive invocation costs $n+1$ shared memory accesses (lines 3–4). Moreover, the sequence of invocations, namely rec_update_snapshot$(n, -)$, rec _update_snapshot$(n-1, -)$, etc., until rec_update_snapshot$(|res|, -)$ (where $x = |res|$ is the recursion level at which the recursion stops) contains $n - |res| + 1$ invocations. It follows that the cost is $O(n(n - |res| + 1))$ shared memory accesses. □

8.6 Summary

This chapter was on the notion of a snapshot object. It has shown how such an object can be wait-free implemented on top of base read/write registers despite asynchrony and any number of process crashes. It has also shown how an implementation for a fixed number of processes can be extended to cope with infinitely many processes. A wait-free implementation of a multi-writer snapshot object has also been described. Finally, the chapter has introduced the notion of an immediate snapshot object and presented both an iterative implementation and a recursive implementation of it.

It is important to insist on the fact that snapshot objects are fundamental objects in crash-prone concurrent environments. This is because they can be used to record the last consistent global state of an application (the value deposited by each process being its last meaningful local state).

8.7 Bibliographic Notes

- Single-writer snapshot objects were introduced independently by Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit [2], and J. Anderson [21].

  The implementation of a single-writer snapshot object described in Fig. 8.6 is due to these authors.

- The best implementation of a single-writer snapshot object known so far requires $O(n \log_2 n)$ accesses to base read/write registers [39]. Finding the corresponding lower bound remains an open problem [93].

- The complexity of update operations of snapshot objects is addressed in [31].

- The extension of the previous implementation to infinitely many processes given in Fig. 8.8 is due to M.K. Aguilera [14].

- Multi-writer snapshot objects were introduced by J. Anderson [22], and M. Inoue, W. Chen, T. Masuzawa and N. Tokura [165].

  Several implementation of multi-writer snapshot objects have been proposed, e.g., [167].

- The implementation of a multi-writer snapshot object described in Fig. 8.10 is due to D. Imbs and M. Raynal [162].

- The notion of partial snapshot was introduced in [36]. A partial snapshot object provides the processes with an operation denoted partial_snapshot($\langle x_1, \ldots, x_y \rangle$), where $\langle x_1, \ldots, x_y \rangle$ denotes the sequence of y components for which the invoking process wants to atomically obtain the values. (If the sequence contains all components, partial_snapshot() boils down to snapshot().) Efficient algorithms that implement partial snapshot objects are described in [158].

- The notion of an immediate snapshot object and its implementation described in Fig. 8.14 are due to E. Borowsky and E. Gafni [53].

  Such objects are central to the definition of iterated computing models [54, 230]. (These distributed computing models are particularly well suited to investigate the computational power of asynchronous systems prone to process crashes.)

- The recursive construction of an immediate snapshot object presented in Sect. 8.5.4 is due to E. Gafni and S. Rajsbaum [112].

8.8 Problem

1. Considering an asynchronous read/write system of n processes where any number of processes may crash, enriched with LL/SC operations (as defined in Sect. 6.3.2), design an efficient partial snapshot implementation of a multi-writer snapshot object.

   Solution in [158].