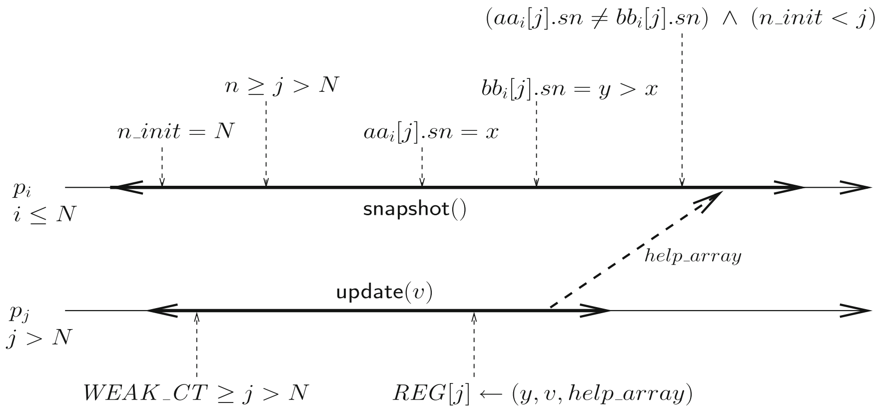

Fig. 8.9 An array transmitted from an update() to a snapshot() operation

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 ypdate(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 $\mathsf{update}(x, v)$ **is**
(1) $sn_i \leftarrow sn_i + 1$;
(2) $REG[x] \leftarrow \langle v, i, sn_i \rangle$;
(3) $HELPSNAP[i] \leftarrow \mathsf{snapshot}()$;
(4) $\mathsf{return}()$
end operation.

operation $\mathsf{snapshot}()$ **is**
(5) $can\_help_i \leftarrow \emptyset$;
(6) **for each** $x \in \{1, \cdots, m\}$ **do** $aa[x] \leftarrow REG[x]$ **end for**;
(7) **repeat forever**
(8) **for each** $x \in \{1, \cdots, m\}$ **do** $bb[x] \leftarrow REG[x]$ **end for**;
(9) **if** $(\forall x \in \{1, \cdots, m\} : aa[x] = bb[x])$ **then** $\mathsf{return}(bb[1..m].val)$ **end if**;
(10) **for each** $x \in \{1, \cdots, m\}$ **such that** $bb[x] \neq aa[x]$ **do**
(11) **let** $\langle -, w, - \rangle = bb[x]$;
(12) **if** $(w \in can\_help_i)$ **then** $\mathsf{return}(HELPSNAP[w])$
(13) **else** $can\_help_i \leftarrow can\_help_i \cup \{w\}$
(14) **end if**
(15) **end for**;
(16) $aa \leftarrow bb$
(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).

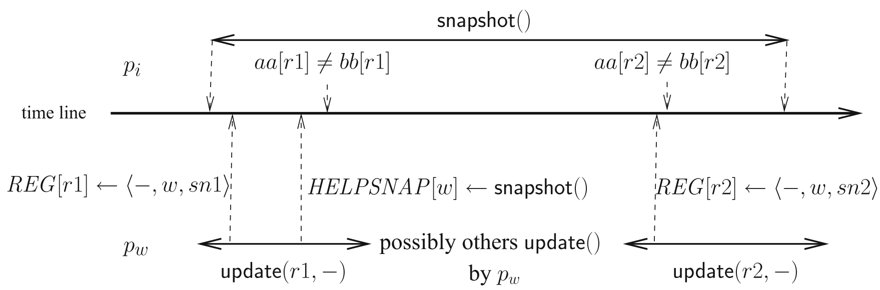

Fig. 8.11 A snapshot() with two concurrent update() by the same process

- 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}).