Adapting to Point Contention with Long-Lived Safe Agreement
(Extended Abstract)

Hagit Attiya

Department of Computer Science
Technion
hagit@cs.technion.ac.il

Abstract. Algorithms with step complexity that depends only on the *point contention*—the number of simultaneously active processes—are very attractive for distributed systems with varying degree of concurrency. Designing shared-memory algorithms that adapt to point contention, using only read and write operations, is however, a challenging task.
The paper specifies the *long-lived safe agreement* object, extending an object of Borowsky et al. [1], and describes an implementation whose step complexity is adaptive to point contention. Then, we illustrate how this object is used to solve other problems, like *renaming* and *information collection*, in an adaptive manner.

1 Introduction

In order to coordinate the actions of a distributed application, processes must obtain up-to-date information from each other. In a typical *wait-free* algorithm, which guarantees that a process completes an operation within a finite number of its own steps, information is collected by reading from an array indexed with process' identifiers. If a distributed algorithm is designed to accommodate a large number of processes, this scheme is an over-kill when only a few processes simultaneously participate in the algorithm: many entries are read from the array although they contain irrelevant information about processes not wishing to coordinate.

The best performance is achieved when the step complexity of an operation is a function only of its *point contention*, namely, the maximal number of processes *simultaneously* executing the algorithm concurrently with it. In this way, an operation is delayed only when many processes are simultaneously active. Note that an algorithm whose step step complexity is *adaptive to point contention* is necessarily wait-free.

For example, if an algorithm is adaptive to point contention, then the step complexity of operation *op* in Figure 1 is constant since at most three processes simultaneously participate at each point during its interval. This holds although a large number of processes are active throughout *op*, and many processes are simultaneously active just before it starts.

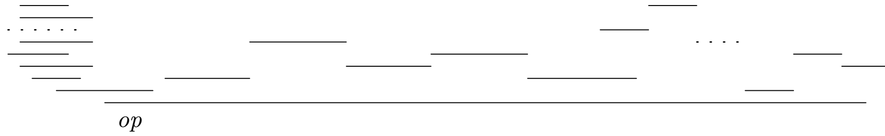

Fig. 1. An execution example.

This article is devoted to explaining the design of algorithms that adapt to point contention using only read and write operations. Our exposition is based on an adaptive implementation of a long-lived safe agreement object.

The *safe agreement* object, originally defined by Borowsky et al. [1], allows processes to propose information and to agree on an identical value. In the BG simulation, the safe agreement object allows processes to agree on each step of a simulated process. Both the specification and the implementation of the object in [1] are neither wait-free (and hence, it is non-adaptive) nor long-lived.

We extend the specification to support adaptive and long-lived properties and present an implementation with $O(k)$ step complexity. Here and below, k denotes the point contention during an operation. This implementation is based on work by Attiya and Zach [2], which in turn follows ideas of Attiya and Fouren [3], as well as Afek et al. [4] and Inoue et al. [5].

The article also describes how safe agreement objects are used to solve the renaming problem and to implement two information collection objects, called *gather* and *collect*.

Renaming [6, 7] allows a process to obtain a *new name*—a positive integer, bounded by a function of the current number of active processes—and to release it afterwards. We adjust a renaming algorithm of Attiya and Fouren [3] to use long-lived safe agreement objects; in the algorithm, processes obtain names in a range of size $O(k^2)$.

Gather and collect objects allow processes to *store* information and to *retrieve* previously-stored information; they differ in the exact properties they provide. Gather and collect objects are commonly used as modular building blocks for efficient adaptive algorithms. We describe an implementation of a gather object, with $O(k^2)$ step complexity for storing information and $O(k)$ step complexity for retrieving information. A collect object is easily implemented from a gather object, with $O(k^2)$ step complexity for storing and retrieving information. The algorithms and their presentation combine ideas from [2, 3, 8, 9].

2 Model of Computation

We use a standard asynchronous shared-memory model of computation, following, e.g., [10]. A system consists of n processes, $p_1, \ldots, p_n$, and a set of registers that are accessed by read and write operations.

An *event* is a computation step by a single process; in an event, a process determines the operation to perform according to its local state, and determines its next local state according to the value returned by the operation.

An *execution* is a (finite or infinite) sequence of events; each event occurs at one process, which applies a read or a write operation to a single register and changes its state according to its algorithm. We assume an *asynchronous* model of computation, where process steps are arbitrarily interleaved.

An invocation of a high-level operation by a process causes the execution of the appropriate algorithm. The *execution interval* of an operation op_i by process p_i is the subsequence of the execution between the first event of p_i in op_i and the last event of p_i in op_i. We denote $op_i \rightarrow op_j$ if the execution interval of op_i precedes the execution interval of the operation op_j by process p_j; namely, the last event of p_i in op_i appears before the first event of p_j in op_j. If neither $op_i \rightarrow op_j$ nor $op_j \rightarrow op_i$, we say that op_i and op_j are *overlapping*.

Let α' be a finite prefix of an execution α; process p_i is *active* at the end of α' if α' includes an invocation of an operation op by p_i without the matching return.

The *point contention* at the end of α', denoted $\mathit{pointCont}(\alpha')$, is the number of active processes at the end of α'. Consider a finite interval β of an execution α; we can write $\alpha = \alpha_1 \beta \alpha_2$. The point contention during β is the maximum contention in prefixes $\alpha_1 \beta'$ of $\alpha_1 \beta$. We abuse notation and denote it by $\mathit{pointCont}(\beta)$, as well; that is

$$\mathit{pointCont}(\beta) = \max\{\mathit{pointCont}(\alpha_1 \beta') : \alpha_1 \beta' \text{ is a prefix of } \alpha_1 \beta \ \} \ .$$

If $\mathit{pointCont}(\beta) = k$, then k processes are simultaneously active at some point during β.

Another measure of concurrency is the *interval contention* during an interval β, denoted $\mathit{intCont}(\beta)$, counting the number of distinct processes that are active at some point during β. Clearly, $\mathit{pointCont}(\beta) \leq \mathit{intCont}(\beta)$.

3 The Adaptive Safe Agreement Object

We start by specifying and implementing the *one-shot* safe agreement object and later extend it to be long-lived.

The original *safe agreement* object [1] allows processes to propose values and to agree on a *single* value; in our safe agreement object, processes agree on an identical *set* of values. Clearly, a single value can be deduced from this set by choosing some predefined value, e.g., the minimum.

In the original specification, a process may wait indefinitely until the agreement value is decided, making it impossible to have a wait-free implementation. This means that there is no adaptive implementation, either. To admit adaptive implementations, we decouple the proposal of a value from the reading of the agreed set, and allow the reading to return an empty set, indicating that decision was not reached yet.

3.1 The One-Shot Object

The one-shot object provides two operations, propose and read. A process invokes propose at most once, but can invoke read to query the object several times.

A propose(*info*) operation tries to store *info* into the object; if it succeeds, it returns **true**, otherwise it returns **false**. A read operation returns a set of values, possibly *empty*.

A one-shot safe agreement object must provide the following properties:

Validity: Any value returned was previously proposed.
Agreement: All non-empty return sets are identical.

A process p_i *accesses* safe agreement object, if it calls a propose operation. A process p_i is *inside* a safe agreement object, if it gets **true** from a propose operation. A process is a *candidate* if its proposed value appears in the non-empty return set of values.

For example, assume processes p_0, p_1, p_2 and p_3 access a safe agreement object (we ignore the specific values proposed), and that p_3 returned **false**, while all other processes returned **true**, and thus, are inside the object. In later read operations, p_0 returns $\emptyset$, while p_1, p_2 and p_3 return $\{p_0, p_1, p_2\}$. In this case, the candidates are p_0, p_1, and p_2.

The *liveness* property, defined next, implies that at least one of the processes that get inside the safe agreement object is guaranteed to recognize itself as a candidate. These processes are called *winners*, and are necessarily also candidates. In the above example, the winners are p_1 and p_2, since they return a non-empty set containing themselves.

Liveness: In any execution, at least one of the processes accessing the object becomes a winner.
Concurrency: If a process gets inside a safe agreement object, and does not win, then some other process is inside the object concurrently.

The adaptive implementation of a one-shot safe agreement object is based on the *sieve* object of Attiya and Fouren [3]. We use the following data structures:

- A Boolean variable *inside*, initially **false**, indicates whether some process is already inside the object.
- An array $R[1 \dots n]$ of views; all views are initially empty. $R[p_i]$ contains the view obtained by process p_i in this object.

We also associate a procedure for one-shot *atomic snapshot* [11] with the object, called osSnap. A process calls the procedure with *info* and obtains a *view*, namely a set of process id's and their information, which must include the process itself. The views returned by the procedure must be *comparable* by containment.

The pseudo-code appears in Algorithm 1.

In a propose operation, a process first checks if it is among the first processes to propose values. It succeeds only if the safe agreement object is empty, and

Algorithm 1 Adaptive one-shot safe agreement object: code for process p_i.

```
data types:
    view : vector of ⟨id, info⟩
shared variables:
    inside : Boolean, initially false
    R[1...n] : array of views, initially ∅
local variables:
    V, W : view

Boolean procedure propose(info)
1:  if (not inside) then
2:      inside = true                  // notify that a process is inside the object
3:      V = osSnap(info)               // propose info; return view of ⟨id, info⟩
4:      R[id_i] = V                    // save the obtained view
5:      return(true)                   // object is open
6:  else return(false)                 // object is not open

view procedure read()                  // returns the subset of proposed values
7:  V = R[id_i]
8:  W = min{R[id_j] | ⟨id_j, *⟩ ∈ V and R[id_j] ≠ ∅}     // min by containment
9:  if ∀⟨id_j, *⟩ ∈ W, R[id_j] ⊇ W then
10:     return(W)
11: else return(∅)
```

inside is **false**. In this case, the process indicates that the safe agreement object is no longer empty, and then obtains a snapshot view, which it stores in its entry in R. The process returns with **true** from propose. If the process does not get inside the object (i.e., *inside* is **true**), then some concurrent process is already inside the object, and the process returns with **false** from propose.

In a read operation, process p_i tries to distinguish the minimal (by containment) snapshot view. This is done by considering the minimal view written by any process in the view obtained by p_i; if one of these processes has not written its view yet, due to the asynchrony of the system, then the read is inconclusive and returns an empty set. Otherwise, the process returns the minimal view it finds. As we shall see in the first lemma below, this suffices to guarantee that all processes return the same set.

The *validity* property of the safe agreement object trivially holds by the code (see [3]).

We argue that all non-empty sets are identical.

Lemma 1. *Algorithm 1 satisfies the* agreement *property.*

Proof. Let V be the minimal view (by containment) obtained in osSnap, by some process p_k. We argue that any non-empty view W returned in an invocation of read by some process p_i is equal to V. Assume, by way of contradiction, that $W \neq V$. Since W is non-empty, it was obtained in some invocation of osSnap; therefore, W and V are comparable. By the minimality of V, $V \subset W$.

Since $p_k \in V \subset W$, p_i checks $R[id_k]$. If $R[id_k]$ is empty then p_i clearly fails the test; otherwise, $R[id_k] = V$ and since $V \subset W$, p_i also fails the test. In both cases, p_i returns $\emptyset$, which is contradiction. ☐

Next, we show that at least one of the processes accessing a safe agreement object becomes a winner, when it calls read after returning from propose.

Lemma 2. *Algorithm 1 satisfies the* liveness *property.*

Proof. Let W_c be the set of candidates in the safe agreement object. W_c is not empty since some processes access the object. Let p_i be the last process in W_c to write its view in propose; clearly, p_i calls read after all processes in W_c write their views in propose.

Let V_i be the view obtained by p_i from osSnap. Since W_c is not empty, some process p_c obtains W_c, that is, the minimal (by containment) view, from osSnap. Note that p_c and p_i could be the same process.

Since W_c and V_i are comparable and since W_c is minimal, $p_c \in W_c \subseteq V_i$. By assumption, p_c writes its view to $R[id_c]$ before p_i calls read. Therefore, p_i reads W_c from $R[id_c]$, and sets V_i to W_c, by its minimality.

By assumption, p_i calls read after every process $p_j \in V_i = W_c$ wrote its view V_j to $R[id_j]$. By the minimality of W_c, $V_i = W_c \subseteq V_j$, and therefore, p_i evaluates the if condition to hold and obtains W_c. Thus, p_i finds that it is a winner in the safe agreement object. ☐

A process that gets inside the safe agreement object first reads **false** from *inside* and then sets *inside* to **true**. This fact can be used to prove that processes are *simultaneously* inside a safe agreement object.

Lemma 3. *The intervals of all processes that get inside a safe agreement object are overlapping.*

The next lemma shows that if process p_i gets inside safe agreement object and does not win, then some process p_j with an overlapping interval is a candidate in the object.

Lemma 4. *Algorithm 1 satisfies the* concurrency *property.*

Sketch of proof. If p_i gets **true** from open then p_i writes **true** to *inside*. By Lemma 3, the intervals of all processes that write **true** to *inside* overlap. By Lemma 2, at least one of them is a winner (and hence, a candidate) the safe agreement object, and the lemma follows. ☐

To evaluate the step complexity of the algorithm, suppose that p_i accesses safe agreement object in interval β_i, and let k be the point contention during β_i.

If A is the set of processes that are inside the object throughout the execution (including p_i itself), then Lemma 3 implies that processes in A access the object concurrently, and hence, $|A| \leq \mathit{pointCont}(\beta_i) = k$. Thus, at most k processes invoke the one-shot snapshot operation, which dominates the step complexity of propose.

Attiya and Fouren [12] present a one-shot snapshot algorithm, whose step complexity is $O(KlogK)$, where K is the total number of processes that ever invoke it (also known as the *total contention*). As we have just argued, in our case we have that $K = k$, the point contention, and thus, the step complexity of the one-shot snapshot algorithm is $O(k \log k)$.

Inoue et al. [5] observed that the one-shot snapshot can be replaced with a *partial atomic snapshot* that guarantees that if one or more processes access the object concurrently, at least one process obtains a snapshot (the others may obtain an empty view). They also present a partial atomic snapshot algorithm with $O(k)$ step complexity, implying that the step complexity of propose is $O(k)$.

Since the maximal size of a view is k, the step complexity of read is $O(k)$. Therefore, a process performs $O(k)$ operations in each invocation of the one-shot safe agreement procedures.

This yields the following theorem:

Theorem 1. *Algorithm 1 implements an adaptive one-shot safe agreement object with $O(k)$ step complexity, where k is the point contention during the operation's execution interval.*

3.2 The Long-lived Object

The long-lived safe agreement object has an infinite number of *generations*. Its interface is similar to that of the one-shot object, except that a *generation number* is added as a parameter to the operations. The generation number is visible from outside the object, in order to simplify the specification of the long-lived object, and to facilitate applications that use the generation number, e.g., the timestamps algorithm of [3].

Specifically, a long-lived safe agreement object supports three operations: propose(*info*) tries to store information in the current generation; if it succeeds, it returns $\langle$**true**,c$\rangle$, otherwise, it returns $\langle$**false**,c$\rangle$. read(c) returns either a non-empty set of values or an empty set. release(c) leaves generation c and activates the next generation $c + 1$, if possible.

A generation starts when the first invocation of a propose operation that returns $\langle *,c \rangle$ occurs, and ends when the last process that got inside this generation leaves the generation, by invoking release(c). I_c denotes the execution interval of generation c.

We assume that there is at most one invocation of propose by each process for each generation, which means that after calling propose that returns a generation number c, the process must call release(c), before calling propose again.

The properties of the one-shot object (validity, agreement, liveness and concurrency) must hold for every generation of the long-lived safe agreement object. We also require the following property:

Synchronization: The generation number is a non-decreasing counter and it is incremented by one during interval I_c. Moreover, processes get inside generation c only after all candidates leave the smaller generations and the generation number is set to c (see Figure 2).

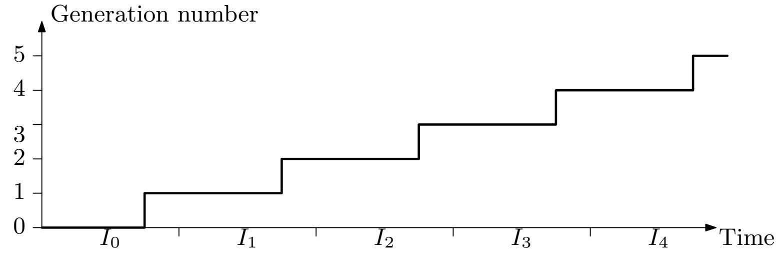

Fig. 2. Generation numbers are nondecreasing.

A long-lived safe agreement object is implemented using an unbounded array of one-shot safe agreement objects, each corresponding to a single generation. An integer variable, *count*, indicates the current generation and points to the current one-shot object. There is also an array $done[1 \ldots n][0 \ldots \infty]$ of Boolean variables, all initially **false**. The entry $done[p_i][c]$ indicates whether process p_i is done with the c'th one-shot safe agreement object. Boolean variables $allDone[0 \ldots \infty]$, all initially **false**, indicate whether all candidates of a specific generation are done.

The pseudo-code appears in Algorithm 2.

As in the one-shot object, a process proposes its value by checking if it is among the first processes to access the current generation of the safe agreement object. The process succeeds only if this generation is *empty*, that is, no other process is already inside it. If the process does not succeed to get inside the current generation, then some concurrent process is already inside the current generation.

The key idea in proving the correctness of the algorithm is that different generations of the same safe agreement object are accessed in disjoint intervals (this is the synchronization property of the object). This property allows correct handling of the non-decreasing counter, and implies that the long-lived safe agreement object inherits the properties of the one-shot object.

Process p_i is inside generation c of a long-lived safe agreement object since it gets **true** from a propose operation of generation c and until it leaves the generation. In other words, process p_i is inside generation c when it is inside the one-shot object $S[c]$ corresponding to generation c.

Careful inspection of the code, and agreement on the set of candidates show that a safe agreement object is released when all candidates are done.

Lemma 5. *If allDone[c] is set to* **true***, then all candidates of S[c] invoked* release*(c).*

This lemma is the key for showing, by induction, that a process is inside generation c only after all candidates leave smaller generations.

Lemma 6. *If process p_i is inside generation c of a safe agreement object, then all the candidates already called* release *of the smaller generations, $1, \ldots, c-1$.*

Algorithm 2 Adaptive long-lived safe agreement object: code for process p_i.

```
shared variables:
    count : integer, initially 1
    S[0...∞] : infinite array of one-shot safe agreement objects
    done[1...n][0...∞] : two-dimensional array of Boolean variables, all initially false
    allDone[0...∞] : infinite array of Boolean variables, all initially false
local variables:
    V, W : view

⟨Boolean, integer⟩ procedure propose(info)
1:  if (all-done[count − 1] and // all candidates of the previous generation are done
          S[count].propose(info)) // and no process is inside the current generation
2:      return(⟨true, count⟩)                      // generation count is open
3:  else release(count)                             // generation c is not open
4:      return(⟨false, count⟩)

view procedure read(c)                   // returns the candidates of generation c
5:  return (S[c].read())

void procedure release(c)                                // release generation c
6:  done[id_i][c] = true                 // indicate that p_i is done in generation c
7:  W = S[c].read()                           // re-calculate the set of candidates
8:  if (p_i ∈ W) then count =c+1                // p_i is a winner in generation c
9:  if (W ≠ ∅ and ∀p_j ∈ W, done[p_j] == true) then       // all candidates are done
10:     allDone[c] = true
```

Sketch of proof. We concentrate on the induction step, assuming that the lemma holds for generation c. Only a winner p_j in generation c writes $c+1$ to *count*, implying that p_j is inside generation c. By the induction hypothesis, all candidates already called release of generation $1, \ldots, c-1$. Since p_i is inside generation $c+1$ it reads **true** from $S[c].allDone$. Lemma 5 implies that all candidates of the one-shot object $S[c]$ already called release on this object. □

Recall that I_c is the execution interval of generation c, starting with the first invocation of a propose operation in generation c and ending when the last process that got inside generation c leaves. The next lemma proves that the data structures of different generations of the same long-lived safe agreement object are modified in disjoint intervals.

Lemma 7. *For every $c \geq 1$, I_c starts after all previous intervals $I_1, \ldots, I_{c-1}$ end.*

Sketch of proof. We concentrate on the induction step, assuming that the lemma holds for for I_c. By definition, I_{c+1} starts with first invocation of a propose operation in generation $c+1$. Lemma 6 implies that if this invocation occurs, then all the candidates already called release of the smaller generations. In particular, all candidates already called release of generation c, and by the induction hypothesis all previous intervals $I_1, \ldots, I_{c-1}$ end.

Since all candidates of generation c already called release, I_c also ended, implying that I_{c+1} starts after all previous intervals end. □

The next lemma shows that the generation number is non-decreasing value, and it is incremented by one during interval I_c.

Lemma 8. *Algorithm 2 satisfies the* synchronization *property.*

Sketch of proof. By the code, only a winner p_w of generation c of a long-lived agreement object writes $c+1$ to *count*. Since p_w is a winner, it previously writes **true** to *inside* of generation c, after I_c begins. Hence, p_w writes $c+1$ to *count* after I_c begins, and before it leaves generation c. Lemma 5 implies that a one-shot safe agreement object is released after all candidates leave the object, and hence, p_w writes $c+1$ to *count* before I_c ends.

Lemma 7 implies that processes get inside generation c only during interval I_c, after *count* is set to c. □

Thus, distinct one-shot objects, associated with different generations, are accessed in disjoint intervals, and the step complexity of accessing the object is dominated by the step complexity of the one-shot object.

Theorem 2. *Algorithm 2 implements an adaptive long-lived safe agreement object with $O(k)$ step complexity, where k is the point contention during the operation's execution interval.*

4 Application I: Renaming

In the (long-lived) renaming problem [6, 7] processes repeatedly *get* and *release* names from a small range. Ideally, the size of the name range should depend only on the current contention.

Long-lived safe agreement objects can be used in a simple renaming algorithm: Place n long-lived safe agreement objects, $S_{LL}[1], \ldots, S_{LL}[n]$, in a row. To get a name, a process p_i accesses the objects sequentially, applying propose to safe agreement object s; if it obtains $\langle \mathbf{true}, c \rangle$, then p_i invokes read(c) to get the set of candidates W. Process p_i returns as its new name the pair composed of s and its rank in W. If p_i obtains $\langle \mathbf{false}, c \rangle$, then it releases $S_{LL}[s]$ and continues to the next object, $S_{LL}[s+1]$. In the latter case, we say that p_i *skips* the s'th safe agreement object.

To release a name, process p_i calls release on the safe agreement object where it obtained a name.

The concurrency properties of the long-lived safe agreement object can easily be used to prove that a process wins within k' iterations, where k' is the interval contention while p_i is getting a name.

It is more surprising—and harder to prove!—that in fact, process p_i wins within $2k-1$ iterations, where k is the point contention while p_i is getting a name. This is done using an interesting *potential* method, which considers two

sets of *simultaneously* active processes and shows that at least one of them is increased by 1 when p_i skips a safe agreement object.

The sets are indexed with an integer number $s = 1, \ldots, n$. The first set, denoted A_s, contains all processes that *access* the ℓ'th safe agreement object, $1 \leq \ell \leq s$. The second set, denote W_s, contains all processes that *win* the ℓ'th safe agreement object, $1 \leq \ell \leq s$. (A_s and W_s are not disjoint.)

The exact definitions of A_s and W_s depend on specific execution intervals, and are used in proving that if process p_i skips the s'th safe agreement object, then $|W_s| + |A_s| \geq s + 1$ (for appropriately chosen intervals). The detailed definitions and statements, as well as the proofs, appear in [3, Section 4].

Since W_s and A_s contain simultaneously active processes, and since p_i is not in W_s, we have that $|W_s| + |A_s| \leq (k-1) + k$. This implies that p_i skips at most $2k-2$ safe agreement objects, and the first component of the new name is $\leq 2k-1$. The second component of the new name is equal to the rank of the process in the obtained view, and thus it is $\leq k$. It follows that the size of the name space is in $O(k^2)$.

Moreover, the step complexity of getting a name is $O(k)$ times the step complexity of proposing and reading in a safe agreement object, since p_i accesses at most $2k-1$ sieves. The step complexity of releasing a name is proportional to the step complexity of releasing a safe agreement object.

Since both complexities can be bound by $O(k)$ (see the previous section), it follows that the algorithm solves long-lived $O(k^2)$-renaming with $O(k^2)$ step complexity.

5 Application II: Collecting Information

We discuss two objects, gather and collect, which allow a process to store its value in a shared memory, or to retrieve the values stored in the shared memory.

The *gather* object provides two operations: a put operation stores its parameter in the object, while a gather operation returns the set of values stored in the object before or possibly during the operation. (This is called the *validity* property.)

The *collect* object provides a collect operation that inherits the validity property of the gather operation, and further guarantees the *regularity* property. Namely, later collect operations are at least as updated as previous ones.

5.1 Implementing a Gather Object

To implement a gather object, we again put n long-lived safe agreement objects, $S_{LL}[1], \ldots, S_{LL}[n]$, in a row. With each safe agreement object we associate a set containing the values of processes that were candidates in this object. In order to store its information, a process iteratively tries to win in the safe agreement objects.

In the first part of a put operation, a process goes through the row of safe agreement objects, until it wins one of them. If it wins safe agreement object

$S_{LL}[s]$, the process merges the values of all candidates of the current generation to the set associated with $S_{LL}[s]$. The synchronization and agreement properties of the safe agreement object guarantee that this set contains the values of all candidates in all generations of $S_{LL}[s]$.

In principle, in a gather operation, the process merely has to go through the safe agreement objects and read the associated sets of values. This might cause a problem, however, if a gather operation is performed after store operations with high contention: A put operation encountering high contention may store a new value in an entry with large index, and a subsequent gather operation will have to access many entries, even if its point contention is low.

This is solved by a technique called *bubble-up*, in which processes propagate new results to the beginning of the row. This allows a gather operation to find the new values at the beginning of the row, when contention is low. Bubble-up was presented by Afek et al. [8] and later used in [2, 3, 9].

After storing its value, the put operation bubbles the result to the top of the array. The process goes from entry s up to entry 1, and for each entry $s' = s, \ldots, 1$, it recursively gathers the values stored beyond entry s', and stores the result in its private register associated with entry s'. (See the description by Stupp [13, Section 7.1].)

In a gather operation, a process starts from the beginning of the row and reads until it find a value that was bubbled up by one of the processes. The number of entries accessed is linear in the point contention during the operation: if the contention is low, then the process reads the value near the beginning of the row; if the contention is high, then the process is allowed to go further in the row.

The key to showing that the gather object is correctly implemented is to show that the result array contains the most recent values of all candidates of all generations of the corresponding safe agreement object. The synchronization property of the safe agreement object is used to show that this property holds, although several processes (winners of the same safe agreement object) may concurrently write to the result entry. (See [3, Lemma 6.1].)

Next we explain how bubbling up restricts the step complexity of the gather operation. Assume that a gather operation by process p_i reads *skips* over entry s. The key to the complexity analysis is to show that a process skips an entry only if some concurrent process is bubbling up through this entry. By careful inspection of the intervals, this is used to show that p_i skips at most $3k$ entries, where k is the point contention.

The same potential argument used for renaming can be used to show that in its first part, a put operation accesses at most $2k - 1$ safe agreement objects. Together with the linear step complexity (in point contention) of operations on a safe agreement object, this implies that the step complexity of a put operation is $O(k^2)$.

5.2 Implementing a Collect Object

When a gather operation returns a value v for process p, it is still possible that a later gather operation returns a value that was written by process p *before* v.

This can happen, for example, when the two gather operations are concurrent with the update operation of process p.

The regularity property of the collect object disallows this behavior. In order to guarantee this property, the collect procedure first calls gather and then calls put to store the result it has obtained. This increases the step complexity of a collect operation to be $O(k^2)$.

6 Summary

We tried to provide a closer look at the algorithmic ideas that are employed in order to have step complexity that adapts to point contention.

This article is not a comprehensive survey of the recent research on adaptive algorithms. The issues not covered here include the space complexity of adaptive algorithms [14,15], using stronger base objects [16,17], guaranteeing weaker types of adaptivity [12,18,19], and adaptive algorithms for other problems, e.g., mutual exclusion [20–22].

References

1. Borowsky, E., Gafni, E., Lynch, N., Rajsbaum, S.: The BG distributed simulation algorithm. Distributed Computing **14**(3) (2001) 127–146
2. Attiya, H., Zach, I.: Fully adaptive algorithms for atomic and immediate snapshots. www.cs.technion.ac.il/∼hagit/pubs/AZ03.pdf (2003)
3. Attiya, H., Fouren, A.: Algorithms adaptive to point contention. Journal of the ACM **50**(4) (2003) 444–468
4. Afek, Y., Attiya, H., Fouren, A., Stupp, G., Touitou, D.: Adaptive long-lived renaming using bounded memory. www.cs.technion.ac.il/∼hagit/pubs/AAFST99disc.ps.gz (1999)
5. Inoue, M., Umetani, S., Masuzawa, T., Fujiwara, H.: Adaptive long-lived $O(k^2)$-renaming with $O(k^2)$ steps. In: Proceedings of the 15th International Conference on Distributed Computing, Berlin, Springer-Verlag (2001) 123–135
6. Attiya, H., Bar-Noy, A., Dolev, D., Peleg, D., Reischuk, R.: Renaming in an asynchronous environment. Journal of the ACM **37**(3) (1990) 524–548
7. Moir, M., Anderson, J.H.: Wait-free algorithms for fast, long-lived renaming. Science of Computer Programming **25**(1) (1995) 1–39
8. Afek, Y., Stupp, G., Touitou, D.: Long-lived and adaptive collect with applications. In: Proceedings of the 40th IEEE Symposium on Foundations of Computer Science, Phoenix, IEEE Computer Society Press (1999) 262–272
9. Afek, Y., Stupp, G., Touitou, D.: Long-lived and adaptive atomic snapshot and immediate snapshot. In: Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing, New-York, ACM Press (2000) 71–80
10. Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems **13**(1) (1991) 124–149
11. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. Journal of the ACM **40**(4) (1993) 873–890
12. Attiya, H., Fouren, A.: Adaptive and efficient algorithms for lattice agreement and renaming. SIAM Journal on Computing **31**(2) (2001) 642–664

13. Stupp, G.: Long Lived and Adaptive Shared Memory Implementations. PhD thesis, Department of Computer Science, Tel-Aviv University (2001)
14. Afek, Y., Boxer, P., Touitou, D.: Bounds on the shared memory requirements for long-lived adaptive objects. In: Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing, New-York, ACM Press (2000) 81–89
15. Attiya, H., Fich, F., Kaplan, Y.: Lower bounds for adaptive collect and related problems. In: Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing. (2004) 60–69
16. Afek, Y., Dauber, D., Touitou, D.: Wait-free made fast. In: Proceedings of the 27th ACM Symposium on Theory of Computing, New-York, ACM Press (1995) 538–547
17. Herlihy, M., Luchangco, V., Moir, M.: Space- and time-adaptive non-blocking algorithms. In: Electronic Notes in Theoretical Computer Science. Volume 78., Elsevier (2003)
18. Afek, Y., Stupp, G., Touitou, D.: Long-lived adaptive splitter and applications. Distributed Computing **15**(2) (2002) 67–86
19. Attiya, H., Fouren, A., Gafni, E.: An adaptive collect algorithm with applications. Distributed Computing **15**(2) (2002) 87–96
20. Anderson, J., Kim, Y.J.: Adaptive mutual exclusion with local spinning. In: Proceedings of the 14th International Conference on Distributed Computing. (2000)
21. Attiya, H., Bortnikov, V.: Adaptive and efficient mutual exclusion. Distributed Computing **15**(3) (2002) 177–189
22. Choy, M., Singh, A.K.: Adaptive solutions to the mutual exclusion problem. Distributed Computing **8**(1) (1994) 1–17