Group Renaming

Yehuda Afek[1], Iftah Gamzu[1⋆], Irit Levy[1], Michael Merritt[2], and Gadi Taubenfeld[3]

[1] School of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel
{afek,iftgam,levyirit}@tau.ac.il
[2] AT&T Labs, 180 Park Ave., Florham Park, NJ 07932, USA
mischu@research.att.com
[3] The Interdisciplinary Center, P.O. Box 167, Herzliya 46150, Israel
tgadi@idc.ac.il

Abstract. We study the *group renaming* task, which is a natural generalization of the *renaming* task. An instance of this task consists of n processors, partitioned into m groups, each of at most g processors. Each processor knows the name of its group, which is in $\{1, \ldots, M\}$. The task of each processor is to choose a new name for its group such that processors from different groups choose different new names from $\{1, \ldots, \ell\}$, where $\ell < M$. We consider two variants of the problem: a *tight* variant, in which processors of the same group must choose the same new group name, and a *loose* variant, in which processors from the same group may choose different names. Our findings can be briefly summarized as follows:

1. We present an algorithm that solves the tight variant of the problem with $\ell = 2m - 1$ in a system consisting of g-consensus objects and atomic read/write registers. In addition, we prove that it is impossible to solve this problem in a system having only $(g-1)$-consensus objects and atomic read/write registers.
2. We devise an algorithm for the loose variant of the problem that only uses atomic read/write registers, and has $\ell = 3n - \sqrt{n} - 1$. The algorithm also guarantees that the number of different new group names chosen by processors from the same group is at most $\min\{g, 2m, 2\sqrt{n}\}$. Furthermore, we consider the special case when the groups are uniform in size and show that our algorithm is self-adjusting to have $\ell = m(m+1)/2$, when $m < \sqrt{n}$, and $\ell = 3n/2 + m - \sqrt{n}/2 - 1$, otherwise.

1 Introduction

1.1 The group renaming problem

We investigate the *group renaming* task which generalizes the well known *renaming* task [3]. In the original renaming task, each processor starts with a unique identifier taken from a large domain, and the goal of each processor is to select a new unique identifier from a smaller range. Such an identifier can be used, for example, to mark

⋆ Supported by the Binational Science Foundation, by the Israel Science Foundation, and by the European Commission under the Integrated Project QAP funded by the IST directorate as Contract Number 015848.

a memory slot in which the processor may publish information in its possession. In the *group renaming* task, groups of processors may hold some information which they would like to publish, preferably using a common memory slot for each group. An additional motivation for studying the group version of the problem is to further our understanding about the inherent difficulties in solving tasks with respect to groups [10].

More formally, an instance of the *group renaming* task consists of n processors partitioned into m groups, each of which consists of at most g processors. Each processor has a *group name* taken from some large name space $[M] = \{1, \ldots, M\}$, representing the group that the processor affiliates with. In addition, every processor has a unique *identifier* taken from $[N]$. The objective of each processor is to choose a new group name from $[\ell]$, where $\ell < M$. The collection of new group names selected by the processors must satisfy the *uniqueness* property meaning that any two processors from different groups choose distinct new group names. We consider two variants of the problem:

- a *tight* variant, in which in addition to satisfying the uniqueness property, processors of the same group must choose the same new group name (this requirement is called the *consistency* property), and
- a *loose* variant, in which processors from the same group may choose different names, rather than a single one, as long as no two processors from different groups choose the same new name.

1.2 Summary of results

We present a wait-free algorithm that solves the tight variant of the problem with $\ell = 2m - 1$ in a system equipped with g-consensus objects and atomic read/write registers. This algorithm extends the upper bound result of Attiya et al. [3] for $g = 1$. On the lower bound side, we show that there is no wait-free implementation of tight group renaming in a system equipped with $(g-1)$-consensus objects and atomic read/write registers. In particular, this result implies that there is no wait-free implementation of tight group renaming using only atomic read/write registers for $g \geq 2$.

We then restrict our attention to shared memory systems which support only atomic read/write reagisters and study the loose variant. We develop a self-adjusting algorithm, namely, an algorithm that achieves distinctive performance guarantees conditioned on the number of groups and processors. On worst case, this algorithm has $\ell = 3n - \sqrt{n} - 1$, while guaranteeing that the number of different new group names chosen by processors from the same group is at most $\min\{g, 2m, 2\sqrt{n}\}$. It seems worthy to note that the algorithm is built around a filtering technique that overcomes scenarios in which both the size of the maximal group and the number of groups are large, i.e., $g = \Omega(n)$ and $m = \Omega(n)$. Essentially, such scenario arises when there are $\Omega(n)$ groups containing only few members and few groups containing $\Omega(n)$ members.

We also consider the special case when the groups are uniform in size, and refine the analysis of our loose group renaming algorithm. Notably, we demonstrate that $\ell = m(m+1)/2$, when $m < \sqrt{n}$, and $\ell = 3n/2 + m - \sqrt{n}/2 - 1$, otherwise. This last result settles, to some extent, an open question posed by Gafni [10].

1.3 Related work

Group solvability was first introduced and investigated in [10]. The renaming problem was first solved for message-passing systems [3], and then for shared memory systems using atomic registers [6]. Both these papers present one-shot algorithms (i.e., solutions that can be used only once). In [8] the first long-lived renaming algorithm was presented: The ℓ-assignment algorithm presented in [8] can be used as an optimal long-lived $(2n-1)$-renaming algorithm with exponential step complexity. Several of the many papers on renaming using atomic registers are [1, 2, 4, 11, 14, 15]. Other references are mentioned later in the paper.

2 Model and Definitions

Our model of computation consists of an asynchronous collection of n processors communicating via shared objects. Each object has a *type* which defines the set of operations that the object supports. Each object also has *sequential specification* that specifies how the object behaves when these operations are applied sequentially. Asynchrony means that there is no assumptions on the relative speeds of the processors.

Various systems differ in the level of *atomicity* that is supported. Atomic (or indivisible) operations are defined as operations whose execution is not interfered with by other concurrent activities. This definition of atomicity is too restrictive, and it is safe to relax it by assuming that processors can try to access the object at the same time, however, although operations of concurrent processors may overlap, each operation should appear to take effect instantaneously. In particular, operations that do not overlap should take effect in there "real-time" order. This type of correctness requirement is called *linearizability* [13].

We will always assume that the system supports *atomic registers*, which are shared objects that support atomic reads and writes operations. In addition, the system may also support forms of atomicity which are stronger than atomic reads and writes. One specific atomic object that will play an important role in our investigation is the consensus object. A consensus object o supports one operation: $o.propose(v)$, satisfying:

1. *Agreement.* In any run, the $o.propose()$ operation returns the same value, called the *consensus value*, to every processor that invokes it.
2. *Validity.* In any run, if the consensus value is v, then some processor invoked $o.propose(v)$.

Throughout the paper, we will use the notation g-consensus to denote a consensus object for g processors.

An object is *wait-free* if it guarantees that *every* processor is always able to complete its pending operation in a finite number of its own steps regardless of the execution speed of other processors (does not admit starvation). Similarly, an implementation or an algorithm is wait-free, if every processor makes a decision within a finite number of its own steps. We will focus only on wait-free objects, implementations or algorithms. Next, we define two notions for measuring the relative computational power of shared objects.

– The *consensus number* of an object of type o, is the largest n for which it is possible to implement an n-consensus object in a wait-free manner, using any number of objects of type o and any number of atomic registers. If no largest n exists, the consensus number of o is infinite.
– The *consensus hierarchy* (also called the wait-free hierarchy) is an infinite hierarchy of objects such that the objects at level i of the hierarchy are exactly those objects with consensus number i.

It has been shown in [12], that in the consensus hierarchy, for any positive i, in a system with i processors: (1) no object at level less than i together with atomic registers can implement any object at level i; and (2) each object at level i together with atomic registers can implement any object at level i or at a lower level, in a system with i processors. Classifying objects by their consensus numbers is a powerful technique for understanding the relative power of shared objects.

Finally, for simplicity, when refereing to the group renaming problem, we will assume that m, the number of groups, is greater or equal to 2.

3 Tight Group Renaming

3.1 An upper bound

In what follows, we present a wait-free algorithm that solves tight group renaming using g-consensus objects and atomic registers. Essentially, we prove the following theorem.

Theorem 1. *For any $g \geq 1$, there is a wait-free implementation of tight group renaming with $\ell = 2m - 1$ in a system consisting of g-consensus objects and atomic registers.*

Corollary 2. *The consensus number of tight group renaming is at most g.*

Our implementation, i.e., Algorithm 1, is inspired by the renaming algorithm of Attiya et al. [3], which achieves an optimal new names space size of $2n - 1$. In this renaming algorithm, each processor iteratively picks some name and suggests it as its new name until an agreement on the collection of new names is reached. The communication between the processors is done using an atomic snapshot object. Our algorithm deviates from this scheme by adding an agreement step between processors of the same group, implemented using g-consensus objects. Intuitively, this agreement step ensures that all the processors of any group will follow the decisions made by the "fastest" processor in the group. Consequently, the selection of the new group names can be determined between the representatives of the groups, i.e., the "fastest" processors. This enables us to obtain the claimed new names space size of $2m - 1$. It is worthy to note that the "fastest" processor of some group may change over time, and hence our agreement step implements a "follow the (current) group leader" strategy. We believe that this concept may be of independent interest. Note that the group name of processor i is designated by GID_i, and the overall number of iterations executed is marked by I.

We now turn to establish Theorem 1. Essentially, this is achieved by demonstrating that Algorithm 1 maintains the consistency and uniqueness properties (Lemmas 4 and

Algorithm 1 Tight group renaming algorithm: code for processor $i \in [N]$.

In shared memory:
$\mathsf{SS}[1, \ldots, N]$ array of swmr registers, initially $\perp$.
$\mathsf{HIS}[1, \ldots, N][1, \ldots, I][1, \ldots, N]$ array of swmr registers, initially $\perp$.
$\mathsf{CON}[1, \ldots, M][1, \ldots, I]$ array of g-consensus objects.

1: $p \leftarrow 1$
2: $k \leftarrow 1$
3: **while** true **do**
4: $\quad \mathsf{SS}[i] \leftarrow \langle \mathsf{GID}_i, p, k \rangle$
5: $\quad \mathsf{HIS}[i][k][1, \ldots, N] \leftarrow$ Snapshot(SS)
$\quad \triangleright$ *Agree on w, the winner of group GID_i in iteration k, and import its snapshot:*
6: $\quad w \leftarrow \mathsf{CON}[\mathsf{GID}_i][k]$.Compete($i$)
7: $\quad (\langle \mathsf{GID}_1, p_1, k_1 \rangle, \ldots, \langle \mathsf{GID}_N, p_N, k_N \rangle) \leftarrow \mathsf{HIS}[w][k][1, \ldots, N]$
$\quad \triangleright$ *Check if p_w, the proposal of w, can be chosen as the new name of group GID_i:*
8: $\quad P = \{p_j : j \in [N]$ has $\mathsf{GID}_j \neq \mathsf{GID}_w$ and $k_j = \max_{q \in [N]}\{k_q : \mathsf{GID}_q = \mathsf{GID}_j\}\}$
9: $\quad$ **if** $p_w \in P$ **then**
10: $\quad\quad r \leftarrow$ the rank of GID_w in $\{\mathsf{GID}_j \neq \perp : j \in [N]\}$
11: $\quad\quad p \leftarrow$ the r-th integer not in P
12: $\quad$ **else** return p_w
13: $\quad$ **end if**
14: $\quad k \leftarrow k + 1$
15: **end while**

5), that it has $\ell = 2m - 1$ (Lemma 6), and that it terminates after a finite number of steps (Lemma 7). Let us denote the value of p written to the snapshot array (see line 4) in some iteration as the *proposal value* of the underlying processor in that iteration.

Lemma 3. *The proposal values of processors from the same group is identical in any iteration.*

Proof. Consider some group. One can easily verify that the processors of that group, and in fact all the processors, have an identical proposal value of 1 in the first iteration. Thus, let us consider some iteration $k > 1$ and prove that all these processors have an identical proposal value. Essentially, this is done by claiming that all the processors update their value of p in the preceding iteration in an identical manner. For this purpose, notice that all the processors compete for the same g-consensus object in that iteration, and then import the same snapshot of the processor that won this consensus (see lines 6–7). Consequently, they execute the code in lines 8–13 in an identical manner. In particular, this guarantees that the update of p in line 11 is done exactly alike. □

Lemma 4. *All the processors of the same group choose an identical new group name.*

Proof. The proof of this lemma follows the same line of argumentation presented in the proof of Lemma 3. Again, the key observation is that in each iteration, all the processors of some group compete for the same g-consensus object, and then import the same snapshot. Since the decisions made by the processors in lines 8–13 are solely based on this snapshot, it follows that they are identical. In particular, this ensures that once a

processor chooses a new group name, all the other processors will follow its lead and choose the same name. □

Lemma 5. *No two processors of different groups choose the same new group name.*

Proof. Recall that we know, by Lemma 4, that all the processors of the same group choose an identical new group name. Hence, it is sufficient that we prove that no two groups select the same new name. Assume by way of contradiction that this is not the case, namely, there are two distinct groups $\mathcal{G}$ and $\mathcal{G}'$ that select the same new group name p^*. Let k and k' be the iteration numbers in which the decisions on the new names of $\mathcal{G}$ and $\mathcal{G}'$ are done, and let $w \in \mathcal{G}$ and $w' \in \mathcal{G}'$ be the corresponding processors that won the g-consensus objects in that iterations. Now, consider the snapshot $(\langle \mathsf{GID}_1, p_1, k_1 \rangle, \ldots, \langle \mathsf{GID}_N, p_N, k_N \rangle)$, taken by w in its k-th iteration. One can easily validate that $p_w = p^*$ since w writes its proposed value before taking a snapshot. Similarly, it is clear that $p'_{w'} = p^*$ in the snapshot $(\langle \mathsf{GID}'_1, p'_1, k'_1 \rangle, \ldots, \langle \mathsf{GID}'_N, p'_N, k'_N \rangle)$, taken by w' in its k'-th iteration. By the linearizability property of the atomic snapshot object and without loss of generality, we may assume that snapshot of w was taken before the snapshot of w'. Consequently, w' must have captured the proposal value of w in its snapshot, i.e., $p'_w = p^*$. This implies that p^* appeared in the set P of w'. However, this violates the fact that w' reached the decision step in line 12, a contradiction. □

Lemma 6. *All the new group names are from the range* $[\ell]$*, where* $\ell = 2m - 1$.

Proof. In what follows, we prove that the proposal value of any processor in any iteration is in the range $[\ell]$. Clearly, this proves the lemma as the chosen name of any group is a proposal value of some processor. Consider some processor. It is clear that its first iteration proposal value is in the range $[\ell]$. Thus, let us consider some iteration $k > 1$ and prove that its proposal value is at most $2m-1$. Essentially, this is done by bounding the value of p calculated in line 11 of the preceding iteration. For this purpose, we first claim that the set P consists of at most $m - 1$ values. Notice that P holds the proposal values of processors from at most $m - 1$ groups. Furthermore, observe that for each of those groups, it holds the proposal values of processors having the same maximal iteration counter. This implies, in conjunction with Lemma 3, that for each of those groups, the proposal values of the corresponding processors are identical. Consequently, P consists of at most $m - 1$ distinct values. Now, one can easily verify that the rank of every group calculated in line 10 is at most m. Therefore, the new value of p is no more than $2m - 1$. □

Lemma 7. *Any processor either takes finite number of steps or chooses a new group name.*

Proof. The proof of this theorem is a natural generalization of the termination proof of the renaming algorithm (see, e.g., [5, Sec. 16.3]). Thus, we defer it to the final version of the paper. □

3.2 An impossibility result

In Appendix A.1, we provide an FLP-style proof of the following theorem.

Theorem 8. *For any $g \geq 2$, it is impossible to wait-free implement tight group renaming in a system having $(g-1)$-consensus objects and atomic registers.*

In particular, Theorem 8 implies that there is no wait-free implementation of tight group renaming, even when $g = 2$, using only atomic registers.

4 Loose Group Renaming

In this section, we restrict our attention to shared memory systems which support only atomic registers. By Theorem 8, we know that it is impossible to solve the tight group renaming problem unless we relax our goal. Accordingly, we consider a variant in which processors from the same group may choose a different new group name, as long as the uniqueness property is maintained. The objective in this case is to minimize both the *inner scope* size, which is the upper bound on the number of new group names selected by processors from the same group, and the *outer scope* size, which is the new group names range size. We use the notation, (α, β)-group renaming algorithm to designate an algorithm yielding an inner scope of α and an outer scope of β.

4.1 The non-uniform case

In the following we consider the task when group sizes are not uniform. We present a group renaming algorithm having a worst case inner scope size of $\min\{g, 2m, 2\sqrt{n}\}$ and a worst case outer scope size of $3n - \sqrt{n} - 1$. The algorithm is self-adjusting with respect to the input properties. Namely, it achieves better performance guarantees conditioned on the number of groups and processors. It seems worthy to emphasize that the performance guarantees of our algorithm are not only based on g and m, but also on $\sqrt{n}$, which is crucial in several cases.

The algorithm is built upon a consolidation of two algorithms, denoted as Algorithm 2 and Algorithm 3. Both algorithms are adaptations of previously known renaming methods for groups (see, e.g., [10]). Algorithm 2, which efficiently handles small-sized groups, is a $(g, n+m-1)$-group renaming algorithm, while Algorithm 3, which efficiently attends to small number of groups, is a $(\min\{m, g\}, m(m+1)/2)$-group renaming algorithm.

Theorem 9. *Algorithm 2 is a wait-free $(g, n+m-1)$-group renaming algorithm.*

Proof. The algorithm is very similar to the original renaming algorithm of Attiya et. al. [3]. While there processors select a new name by computing the rank of their original large id among the ids of participating processors, here processors consider the rank of their original *group name* among the already published (participating) original group names. One can prove that Algorithm 2 maintains the uniqueness property and terminates after finite number of steps by applying nearly identical arguments to those used in the analogous proofs of the underlying renaming method (see, e.g., [5, Sec. 16.3]). Therefore, we only focus on analyzing the size of the resulting new name-spaces. The inner scope size of the algorithm is trivially g since there are at most g processors in any group. We turn to bound the outer scope size. This is done by demonstrating that

the proposal value p_i of any processor i in any iteration is at most $n + m - 1$. Clearly, p_i satisfies this requirement in the first iteration as its value is 1. Hence, let us consider some other iteration and bound its proposal value. This is accomplished by bounding the value of p_i calculated in line 7 of the preceding iteration. For this purpose, notice that the rank of every group calculated in line 6 is at most m. Furthermore, there are at most $n - 1$ values proposed by other processors. Thus, the new value of p_i is at most $n + m - 1$. □

Algorithm 2 code for processor $i \in [N]$.

In shared memory: $\mathsf{SS}[1, \ldots, N]$ array of swmr registers, initially $\perp$.

```
1:  p_i ← 1
2:  while true do
3:      SS[i] ← ⟨GID_i, p_i⟩
4:      (⟨GID_1, p_1⟩, ..., ⟨GID_N, p_N⟩) ← Snapshot(SS)
5:      if p_i = p_j for some j ∈ [N] having GID_j ≠ GID_i then
6:          r ← the rank of GID_i in {GID_k ≠ ⊥ : k ∈ [N]}
7:          p_i ← the r-th integer not in {p_k ≠ ⊥ : k ∈ [N] \ {i}}
8:      else return p_i
9:      end if
10: end while
```

Theorem 10. *Algorithm 3 is a wait-free* $(\min\{m, g\}, m(m+1)/2)$*-group renaming algorithm.*

Proof. In this algorithm each processor records its participation by publishing its id and its group original name. Each processor then takes a snapshot of the memory and returns as its new group name the size of the snapshot it had obtained, concatenated with its group id rank among the group ids recorded in the snapshot. One can prove that Algorithm 3 supports the uniqueness property by applying nearly identical arguments to those used in the corresponding proof of the underlying renaming method (see, e.g., [7, Sec. 6]). Moreover, it is clear that the algorithm terminates after finite number of steps. Thus, we only focus on analyzing the performance properties of the algorithm. We begin with the inner scope size. Particularly, we prove a bound of m, noting that a bound of g is trivial since there are at most g processors in any group. Consider the case that two processors of the same group obtain the same number of observable groups $\tilde{m}$ in line 3. We argue that they also choose the same new group name. For this purpose, notice that the set of GIDs that reside in SS may only grow during any execution sequence. Hence, if two processors have an identical $\tilde{m}$ then their snapshot holds the same set of GIDs. Consequently, if those processors are of the same group then their group rank calculated in line 4 is also the same, and therefore the new names they select are identical. This implies that the number of new group names selected by processors from the same group is bound by the maximal value of $\tilde{m}$, which is clearly never greater than m. We continue by bounding the outer scope size. As already noted,

$\tilde{m} \leq m$, and the rank of every group is at most m. Thus, the maximal group name is no more than $m(m-1)/2 + m$. □

Algorithm 3 code for processor $i \in [N]$.

In shared memory: $\mathsf{SS}[1, \ldots, N]$ array of swmr registers, initially $\perp$.

1: $\mathsf{SS}[i] \leftarrow \mathsf{GID}_i$
2: $(\mathsf{GID}_1, \ldots, \mathsf{GID}_N) \leftarrow$ Snapshot(SS)
3: $\tilde{m} \leftarrow$ the number of distinct GIDs in $\{\mathsf{GID}_j \neq \perp : j \in [N]\}$
4: $r \leftarrow$ the rank of GID_i in $\{\mathsf{GID}_j \neq \perp : j \in [N]\}$
5: return $\tilde{m}(\tilde{m}-1)/2 + r$

We are now ready to present our self-adjusting loose group renaming algorithm. The algorithm has its roots in the natural approach that applies the best response with respect to the instance under consideration. For example, it is easy to see that Algorithm 3 outperforms Algorithm 2 with respect to the inner scope size, for any instance. In addition, one can verify that when $m < \sqrt{n}$, Algorithm 3 has an outer scope size of at most $n/2 - \sqrt{n}/2$, whereas Algorithm 2 has an outer scope size of at least n. Hence, given an instance having $m < \sqrt{n}$, the best response would be to execute Algorithm 3. Unfortunately, a straight-forward application of this approach has several difficulties.

One immediate difficulty concerns the implementation since none of the processors have prior knowledge of the real values of m or g. Our algorithm bypasses this difficulty by maintaining an estimation of these parameters using an atomic snapshot object. Another difficulty concerns with performance issues. Specifically, both algorithms have poor inner scope size guarantees for instances which simultaneously satisfy $g = \Omega(n)$ and $m = \Omega(n)$. One concrete example having $g = n/2$ and $m = n/2 + 1$ consists of a single group having $n/2$ members and $n/2$ singleton groups. In this case, both algorithms have an inner scope size guarantee of $n/2$. We overcome this difficulty by sensibly combining the algorithms, therefore yielding an inner scope size guarantee of $2\sqrt{n}$ for these "hard" cases. The key observation utilized in this context is that if there are many groups then most of them must be small. Consequently, by filtering out the small-sized groups, we are left with a small number of large groups that we can handle efficiently. Note that Algorithm 4 employs Algorithm 3 as sub-procedure in two cases (see lines 6 and 12). It is assumed that the shared memory space used by each application of the algorithm is distinct.

Theorem 11. *Algorithm 4 is a group renaming algorithm having a worst case inner scope size of* $\min\{g, 2m, 2\sqrt{n}\}$ *and a worst case outer scope size of* $3n - \sqrt{n} - 1$.

Proof. We begin by establishing the correctness of the algorithm. For this purpose, we demonstrate that it maintains the uniqueness property and terminates after finite number of steps. One can easily validate that the termination property holds since both Algorithm 2 and Algorithm 3 terminate after finite number of steps. It is also easy to verify that the uniqueness property is maintained. This follows by recalling that both

Algorithm 4 Adjusting group renaming algorithm: code for processor $i \in [N]$.

In shared memory: $\mathsf{SS}[1,\ldots,N]$ array of swmr registers, initially $\perp$.

1: $\mathsf{SS}[i] \leftarrow \mathsf{GID}_i$
2: $(\mathsf{GID}_1,\ldots,\mathsf{GID}_N) \leftarrow$ Snapshot(SS)
3: $\tilde{m} \leftarrow$ the number of distinct GIDs in $\{\mathsf{GID}_j \neq \perp : j \in [N]\}$
4: $\tilde{g} \leftarrow$ the number of processors $j \in [N]$ having $\mathsf{GID}_j = \mathsf{GID}_i$
5: **if** $\tilde{m} < \sqrt{n}$ **then**
6: $\quad x \leftarrow$ the outcome of Algorithm 3 (using shared memory $\mathsf{SS}_1[1,\ldots,N]$)
7: $\quad$ return x
8: **else if** $\tilde{g} \leq \sqrt{n}$ **then**
9: $\quad x \leftarrow$ the outcome of Algorithm 2 (using shared memory $\mathsf{SS}_2[1,\ldots,N]$)
10: $\quad$ return $x + n/2 - \sqrt{n}/2$
11: **else**
12: $\quad x \leftarrow$ the outcome of Algorithm 3 (using shared memory $\mathsf{SS}_3[1,\ldots,N]$)
13: $\quad$ return $x + 5n/2 - \sqrt{n}/2 - 1$
14: **end if**

Algorithm 2 and Algorithm 3 maintain the uniqueness property, and noticing that each case of the if statement (see lines 5–14) utilizes a distinct set of new names. To be precise, one should observe that any processor that executes Algorithm 3 in line 6 is assigned a new name in the range $\{1,\ldots,n/2-\sqrt{n}/2\}$, any processor that executes Algorithm 2 in line 9 is assigned a new name in the range $\{n/2-\sqrt{n}/2+1,\ldots,5n/2-\sqrt{n}/2-1\}$, and any processor that executes Algorithm 3 in line 12 is assigned a new name whose value is at least $5n/2-\sqrt{n}/2$. The first claim results by the outer scope properties of Algorithm 3 and the fact that processors from less than $\sqrt{n}$ groups may execute this algorithm. The second argument follows by the outer scope properties of Algorithm 2, combined with the observation that $m \leq n$, and the fact that the value of the name returned by the algorithm is increased by $n/2-\sqrt{n}/2$ in line 10. Finally, the last claim holds since Algorithm 3 is guaranteed to attain a positive-valued integer name, and the value of this name is increased by $5n/2-\sqrt{n}/2-1$ in line 13.

We now turn to establish the performance properties of the algorithm. We demonstrate that it is self-adjusting and has the following (inner scope, outer scope) properties:

$$\begin{cases} (\ \min\{m,g\},\ \ m(m+1)/2\ \ ) & m < \sqrt{n} \\ (\ g,\ \ 3n/2+m-\sqrt{n}/2-1\ \ ) & m \geq \sqrt{n} \text{ and } g \leq \sqrt{n} \\ (\ \min\{g, 2\sqrt{n}\},\ 3n-\sqrt{n}-1\ ) & m \geq \sqrt{n} \text{ and } g > \sqrt{n} \end{cases}$$

Case I: $\mathbf{m < \sqrt{n}}$. The estimation value $\tilde{m}$ always satisfy $\tilde{m} \leq m$. Therefore, all the processors execute Algorithm 3 in line 6. The properties of Algorithm 3 guarantee that the inner scope size is $\min\{m,g\}$ and the outer scope size is $m(m+1)/2$. Take notice that $\min\{m,g\} \leq \min\{g, 2m, 2\sqrt{n}\}$ and $m(m+1)/2 \leq 3n-\sqrt{n}-1$ since $m < \sqrt{n}$. Thus, the performance properties of the algorithm in this case support the worst case analysis.

Case II: $\mathbf{m \geq \sqrt{n}}$ and $\mathbf{g \leq \sqrt{n}}$. The estimation values never exceed their real values, namely, $\tilde{m} \leq m$ and $\tilde{g} \leq g$. Consequently, some processors may execute Algorithm 3 in

line 6 and some may execute Algorithm 2 in line 9, depending on the concrete execution sequence. The inner scope size guarantee is trivially satisfied since there are at most g processors in each group. Furthermore, one can establish the outer scope size guarantee by simply summing the size of the name space that may be used by Algorithm 3, which is $n/2 - \sqrt{n}/2$, with the size of the name space that may be used by Algorithm 2, which is $n + m - 1$. Notice that $g \leq \min\{g, 2m, 2\sqrt{n}\}$ since $g \leq \sqrt{n} \leq m$, and $3n/2 + m - \sqrt{n}/2 - 1 \leq 3n - \sqrt{n} - 1$ as $m \leq n$. Hence, the performance properties of the algorithm in this case support the worst case analysis.

Case III: $\mathbf{m \geq \sqrt{n}}$ and $\mathbf{g > \sqrt{n}}$. Every processors may execute any of the algorithms, depending of the concrete execution sequence. The first observation one should make is that no more than $\sqrt{n}$ new names may be collectively assigned to processors of the same group by Algorithm 3 in line 6 and Algorithm 2 in line 9. Moreover, one should notice that any processor that executes Algorithm 3 in line 12 is part of a group of size greater than $\sqrt{n}$. Consequently, processors from less than $\sqrt{n}$ groups may execute it. This implies, in conjunction with the properties of Algorithm 3, that no more than $\sqrt{n}$ new names may be assigned to each group, and at most $n/2 - \sqrt{n}/2$ names are assigned by this algorithm. Putting everything together, we attain that the inner scope size is $\min\{g, 2\sqrt{n}\}$ and the outer scope size is $3n - \sqrt{n} - 1$. It is easy to see that $\min\{g, 2\sqrt{n}\} \leq \min\{g, 2m, 2\sqrt{n}\}$ since $m \geq \sqrt{n}$, and thus the performance properties of the algorithm in this case also support the worst case analysis. □

4.2 The uniform case

In what follows, we study the problem when the groups are guaranteed to be uniform in size. We refine the analysis of Algorithm 4 by establishing that it is a loose group renaming algorithm having a worst case inner scope size of $\min\{m, g\}$, and an outer scope size of $3n/2 + m - \sqrt{n}/2 - 1$. Note that $\min\{m, g\} \leq \sqrt{n}$ in this case. In particular, we demonstrate that the algorithm is self-adjusting and has the following (inner scope, outer scope) properties:

$$\begin{cases} (\ \min\{m, g\},\ m(m+1)/2\ ) & m < \sqrt{n} \\ (\ g,\ 3n/2 + m - \sqrt{n}/2 - 1\ ) & m \geq \sqrt{n} \end{cases}$$

This result settles, to some extent, an open question posed by Gafni [10], which called for a self-adjusting group renaming algorithm that requires at most $m(m+1)/2$ names on one extreme, and no more than $2n - 1$ names on the other.

The key observation required to establish this refinement is that $n = m \cdot g$ when the groups are uniform in size. Consequently, either $m < \sqrt{n}$ or $g \leq \sqrt{n}$. Since the estimation values that each processor sees cannot exceed the corresponding real values, no processor can ever reach the second execution of Algorithm 3 in line 12. Now, the proof of the performance properties follows the same line of argumentation presented in the proof of Theorem 11.

5 Discussion

This paper has considered and investigated the tight and loose variants of the group renaming problem. Below we discuss few ways in which our results can be extended. An

immediate open question is whether a g-consensus task can be constructed from group renaming tasks for groups of size g, in a system with g processes. Another question is to design an *adaptive* group renaming algorithm in which a processor is assigned a new group name, from the range 1 through k where k is a constant multiple of the contention (i.e., the number of different active groups) that the processor experiences. We have considered only one-shot tasks (i.e., solutions that can be used only once), it would be interesting to design long-lived group renaming algorithms. We have focused in this work mainly on reducing the new name space as much as possible, it would be interesting to construct algorithms also with low space and time (step) complexities. Finally, the *k-set* consensus task, a generalization of the consensus task, enables for each processor that starts with an input value from some domain, to choose some participating processor' input as its output, such that all processors together may choose no more than k distinct output values. It is interesting to find out what type of group renaming task, if any, can be implemented using k-set consensus tasks and registers.

References

1. Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Long-lived renaming made adaptive. In *Proc. 18th ACM Symp. on Principles of Distributed Computing*, pages 91–103, May 1999.
2. Y. Afek, G. Stupp, and D. Touitou. Long lived adaptive splitter and applications. *Distributed Computing*, 30:67–86, 2002.
3. H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3):524–548, 1990.
4. H. Attiya and A. Fouren. Algorithms adapting to point contention. *Journal of the ACM*, 50(4):144–468, 2003.
5. H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley Interscience, 2004.
6. A. Bar-Noy and D. Dolev. Shared memory versus message-passing in an asynchronous. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 307–318, 1989.
7. A. Bar-Noy and D. Dolev. A partial equivalence between shared-memory and message-passing in an asynchronous fail-stop distributed environment. *Mathematical Systems Theory*, 26(1):21–39, 1993.
8. J. Burns and G. Peterson. The ambiguity of choosing. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 145–158, Aug. 1989.
9. M. J. Fischer, N. A. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
10. E. Gafni. Group-solvability. In *Proceedings 18th International Conference on Distributed Computing*, pages 30–40, 2004.
11. E. Gafni, M. Merritt, and G. Taubenfeld. The concurrency hierarchy, and algorithms for unbounded concurrency. In *Proc. 20th ACM Symp. on Principles of Distributed Computing*, pages 161–169, Aug. 2001.
12. M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.
13. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
14. M. Inoue, S. Umetani, T. Masuzawa, and H. Fujiwara. Adaptive long-lived $O(k^2)$-renaming with $O(k^2)$ steps. In *15th international symposium on distributed computing*, 2001. *LNCS 2180* Springer Verlag 2001, 123–135.

15. M. Moir and J. H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1):1–39, Oct. 1995.

A Tight Group Renaming

A.1 An impossibility result

In what follows, we establish the proof of Theorem 8. Our impossibility proof follows the high level FLP-approach employed in the context of the consensus problem (see, e.g., [12, 9]). Namely, we assume the existence of a tight group renaming algorithm, and then derive a contradiction by constructing a sequential execution in which the algorithm fails, either because it is inconsistent, or since it runs forever. Prior to delving into technicalities, we introduce some terminology.

The *decision value* of a processor is the new group name selected by that processor. Analogously, the decision value of a group is the new group name selected by all processors of that group. An algorithm state is *multivalent* with respect to group $\mathcal{G}$ if the decision value of $\mathcal{G}$ is not yet fixed, namely, the current execution can be extended to yield different decision values of $\mathcal{G}$. Otherwise, it is *univalent*. In particular, an *x-valent* state with respect to $\mathcal{G}$ is a univalent state with respect to $\mathcal{G}$ yielding a decision value of x. A *decision step* with respect to $\mathcal{G}$ is an execution step that carries the algorithm from a multivalent state with respect to $\mathcal{G}$ to a univalent state with respect to $\mathcal{G}$. A processor is *active* with respect to a algorithm state if its decision value is still not fixed. A algorithm state is *critical* with respect to $\mathcal{G}$ if it is multivalent with respect to $\mathcal{G}$ and any step of any active processor is a decision step with respect to $\mathcal{G}$.

Lemma 12. *Every group renaming algorithm admits an input instance whose initial algorithm state is multivalent with respect to a maximal size group.*

Proof. We begin by establishing that every group renaming algorithm admits an input instance whose initial algorithm state is multivalent with respect to *some* group. Consider some group renaming algorithm, and assume by contradiction that the initial algorithm state is univalent with respect to all groups for every input instance. We argue that all processors implement some function $f : [M] \to [\ell]$ for computing their new group name. For this purpose, consider some processor whose group name is $a \in [M]$. Notice that this processor may be scheduled to execute a "solo run". Let us assume that its decision value in this case is $x \in [\ell]$. Since the initial algorithm state is univalent with respect to the group of that processor, it follows that in any execution this processor must decide x, regardless of the other groups, their name, and their scheduling. The above-mentioned argument follows by recalling that all processors execute the same algorithm, and noticing that a could have been any initial group name. Now, recall that $M > \ell$. This implies that there are at least two group names $a_1, a_2 \in [M]$ such that $f(a_1) = f(a_2)$. Correspondingly, there are input instances in which two processors from two different groups decide on the same new group name, violating the uniqueness property.

We now turn to prove that every group renaming algorithm admits an input instance whose initial algorithm state is multivalent with respect to a maximal size group. Consider some group renaming algorithm, and suppose its initial algorithm state is multivalent with respect to group $\mathcal{G}$. Namely, there are two execution sequences σ_1, σ_2 that

lead to different decision values of $\mathcal{G}$. Now, if $\mathcal{G}$ is maximal in size then we are done. Otherwise, consider the input instance obtained by adding processors to $\mathcal{G}$ until it becomes maximal in size. Notice that the execution sequences σ_1 and σ_2 are valid with respect to the new input instance. In addition, observe that each possessor must decide on the same value as in the former instance. This follows by the assumption that none of the processors has prior knowledge about the other processors and groups, and thus each processor cannot distinguish between the two instances. Hence, the initial algorithm state is also multivalent with respect to $\mathcal{G}$ in this new instance. □

Lemma 13. *Every group renaming algorithm admits an input instance for which a critical state with respect to a maximal size group may be reached.*

Proof. We prove that every group renaming algorithm which admits an input instance whose initial algorithm state is multivalent with respect to some group may reach a critical state with respect to that group. Notice that having this claim proved, the lemma follows as consequence of Lemma 12. Consider some group renaming algorithm, and suppose its initial algorithm state is multivalent with respect to group $\mathcal{G}$. Consider the following sequential execution, starting from this state. Initially, some arbitrary processor executes until it reaches a state where its next operation leaves the algorithm in a univalent state with respect to $\mathcal{G}$, or until it terminates and decides on a new group name. Note that the latter case can only happen if the underlying processor is not affiliated to $\mathcal{G}$. Also note that the processor must eventually reach one of the above-mentioned states since the algorithm is wait-free and cannot run forever. Later on, another arbitrary processor executes until it reaches a similar state, and so on. This sequential execution continues until reaching a state in which any step of any active processor is a decision step with respect to $\mathcal{G}$. Again, since the algorithm cannot run forever, it must eventually reach such state, which is, by definition, critical. □

We are now ready to prove the impossibility result.

Proof of Theorem 8. Assume that there is a group renaming algorithm implemented from atomic registers and r-consensus objects, where $r < g$. We derive a contradiction by constructing an infinite sequential execution that keeps such algorithm in a multivalent state with respect to some maximal size group. By Lemma 13, we know that there is an input instance and a corresponding execution of the algorithm that leads to a critical state s with respect to some group $\mathcal{G}$ of size g. Keep in mind that there are at least g active processors in this critical state since, in particular, all the processors of $\mathcal{G}$ are active. Let p and q be two active processors in the critical state which respectively carry the algorithm into an x-valent and a y-valent states with respect to $\mathcal{G}$, where x and y are distinct. We now consider four cases, depending on the nature of the decision steps taken by the processors:

Case I: One of the processors reads a register. Let us assume without loss of generality that this processor is p. Let s' be the algorithm state reached if p's read step is immediately followed by q's step, and let s'' be the algorithm state following q's step. Notice that s' and s'' differ only in the internal state of p. Hence, any processor $p' \in \mathcal{G}$, other than p, cannot distinguish between these states. Thus, if it executes a "solo run", it must decide on the same value. However, an impossibility follows since s' is

x-valent with respect to $\mathcal{G}$ whereas s'' is y-valent. This case is schematically described in Figure 1(a).

Case II: Both processors write to the same register. Let s' be the algorithm state reached if p's write step is immediately followed by q's write step, and let s'' be the algorithm state following q's write step. Observe that in the former scenario q overwrites the value written by p. Hence, s' and s'' differ only in the internal state of p. Therefore, any processor $p' \in \mathcal{G}$, other than p, cannot distinguish between these states. The impossibility follows identically to Case I.

Case III: Each processor writes to or competes for a distinct register or consensus object. In what follows, we prove impossibility for the scenario in which both processors write to different registers, noting that impossibility for other scenarios can be easily established using nearly identical arguments. The algorithm state that results if p's write step is immediately followed by q's write step is identical to the state which results if the write steps occur in the opposite order. This is clearly impossible as one state is x-valent and the other is y-valent. This case is schematically illustrated in Figure 1(b).

Case IV: All active processors compete for the same consensus object. As mentioned above, there are at least g active processors in the critical state. Additionally, we assumed that the algorithm uses r-consensus objects, where $r < g$. This implies that the underlying consensus object is accessed by more processors then its capacity, which is illegal. ■

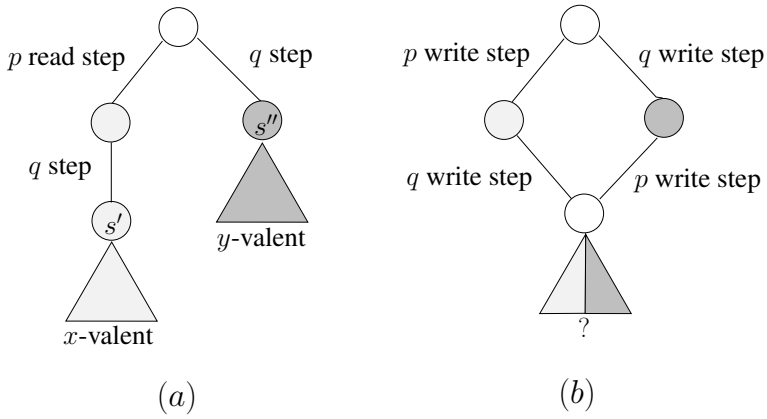

Fig. 1. The decision steps cases.