Tight Group Renaming on Groups of Size g Is Equivalent to g-Consensus

Yehuda Afek[1], Eli Gafni[2], and Opher Lieber[1]

[1] The Blavatnik School of Computer Science, Tel-Aviv University, Israel
[2] Computer Science Department, Univ. of California, LA, CA

Abstract. We address two problems, the g-tight group renaming task and what we call, *safe*-consensus task, and show the relations between them. We show that any g-tight group renaming task, the first problem, implements g processes consensus. We show this by introducing an intermediate task, the *safe*-consensus task, the second problem, and showing that g-tight group renaming implements g-safe-consensus and that the latter implements g-consensus. It is known that with g-consensus g-tight group renaming is solvable, making the two problems equivalent.

The *safe*-consensus task, is of independent interest. In it the validity condition of consensus is weakened as follows: if the first processor to invoke the task returns before any other processor invokes, i.e., it runs in solo, then it outputs its input; Otherwise the consensus output can be arbitrary, not even the input of any process. We show the equivalence between safe-(set-)consensus and (set-)consensus.

Keywords: consensus, validity, set-consensus, group renaming, solo run.

1 Introduction

The notion of group solvability was introduced in [9]. The paper in [3] introduced a simpler version of group solvability called tight group solvability and in particular tackled the task of tight group renaming. The tight group renaming task is the renaming problem [2] of groups. Groups have to agree on a slot, and different groups have to agree on different slots. In the g-tight group renaming task there are g processors in each group. In [3] it was shown that g-consensus is sufficient to solve g-tight group renaming. The question whether it is also necessary was left open.

Here we introduce a new task, *safe-consensus*, a weakening of the classic consensus problem, and show the relation between this task, the g-tight group renaming task and the classic g-consensus task. The two basic conditions satisfied by the consensus problem [8] are, *agreement* and *validity.* In the *safe-consensus* task we weaken the *validity* condition of consensus. In the classic consensus problem the output to a participating processor is the input of a participating processor. In a *safe-consensus* task, the validity is weakened to allow the task to return an arbitrary value if initially the number of participating processors is two or more. The agreement property is retained.

Our first result shows that g-tight group renaming implements g processes safe-consensus. Essentially, safe-consensus is an abstraction of tight group renaming. In any deterministic solution to the tight group renaming the output of a solo run is a function of the processor's id and its group id. Since we do not restrict the algorithm there may be two solo runs from the same group that output different values. Thus if two or more processors from the same group come together any new group name may be output, resembling the behavior of a safe-consensus task.

Our next result shows that safe-consensus is in fact as powerful as consensus, which together with the previous result shows its necessity to solve g-tight group renaming.

Following the introduction of *safe-consensus*, we next examine ways to weaken the validity condition of (n, k)-set-consensus. In (n, k)-set-consensus a processor outputs an input of a participating processor, and the cardinality of the output set is no larger than k. Hence the problem becomes non-trivial when initially more than k processors access the task concurrently. Thus the "off-the-cuff" weakening of the validity condition here is, that if more than k processors participate, it can return default non-valid outputs such that the total number of distinct values returned does not exceed k.

We show that this natural weakening is not equivalent to the original problem. Indeed it is strictly weaker than (n, k)-set-consensus. Nevertheless, it is a non-trivial task - it is not read-write wait-free solvable. We then consider strengthening the validity condition of (n, k)-safe-set-consensus to imitate safe-consensus: If a non-valid output is returned to any processors then it must be returned to all. We then show that strong-safe-set-consensus is equivalent to classic set-consensus.

Related Work: In [10], Guerraoui, and Kuznetsov define the weak consensus task. Here processors output 0 or 1 with the validity requirement being only that there exists a run of the task that outputs 0, and there exists a run that outputs 1. They show how weak consensus for n processors can be used to implement n processor consensus with the standard validity condition. Unlike this paper, they rely on the fact that they deal with an *object*, that is a given deterministic implementation of a task. In fact, they need to drive the implementation into particular special state (treating it as a white box). We on the other hand deal just with the specification, i.e., task (as a black box). We do not rely on any particular implementation of the task.

2 Model and Problem Definitions

We follow the standard model of asynchronous shared memory system as in [12,11]. There are n processors in the system $\{1, 2, \ldots, n\}$ that communicate by either atomically reading and writing to the atomic read/write shared memory, or by applying operations to a shared object such as a consensus object.

The g-tight group renaming problem: We follow the definition from [3]: In a tight group renaming task with group size g, n processors with id's from a large domain $\{1, 2, \ldots, N\}$ are partitioned into m groups with id's from a large domain $\{1, 2, \ldots, M\}$, with at most g processors per group. A tight group renaming task renames groups from the domain $1..M$ to $1..l$ for $l << M$, where all processors with the same initial group ID are renamed to the same new group ID, and no two different initial group id's are renamed to the same new group ID.

The n-Safe-consensus problem: In this task n processors with id's $1..n$ each proposes an input value, and outputs a value such that:

- Wait-Free: Each processor finishes executing within a finite number of steps.
- Agreement: All processors output the same output value.
- Weak-Validity: If the output of a processor occurs before the invocation of any other processor then the output is that processor's proposed input value.

Hence, if no processor initially accesses the Safe-consensus task in solo then processors may agree on any value. Notice, a similar task but in which the agreement condition is that each process may return either a fixed default value or the agreement value is read/write implementable.

The (n, k)-Safe-set-consensus problem: In this task n processors with id's $1..n$ each proposes an input value, and outputs a value such that:

- Wait-Free: Each processor finishes executing within a finite number of steps.
- k-Agreement: At most k distinct values are output.
- Weak-Validity: If the first output occurs after no more than k processors have invoked, then all processors output a proposed input value.

The Strong *(n, k)-Safe-set-consensus problem:* In this task n processors with id's $1..n$ each proposes an input value, and outputs a value such that:

- Wait-Free: Each processor finishes executing within a finite number of steps.
- Strong k-Agreement: At most k distinct values are output, and if any output value is not a proposed input value, that output value is the output of all processors which access the task.
- Weak-Validity: If the first output occurs after no more than k processors have invoked, then all processors output a proposed input value.

Task equivalence: We use wait-free constructions to compare two tasks, A and B. We say that "*A can implement B*" if there is a wait-free algorithm C that may use any number of copies of task A and read/write atomic registers to solve task B. If A and B implement each other, the tasks are said to be *equivalent.*

Atomic Snapshots: In several of the algorithms we utilize the ability to perform atomic snapshots of shared memory, as defined in [1].

Correctness proofs for all the algorithms are given in Appendix A.

3 g-Tight Group Renaming Implements g-Safe-Consensus

We show that a single g-tight group renaming task can implement g-safe-consensus so long as it supports at least $g + 1$ processor invocations. We then show that any weakening of this requirement is impossible.

Theorem 1. *Any tight group renaming task with group size g supporting n processors s.t., $n > g$ and $M > l$ implements 0/1 safe-consensus for g processors.*

Given an algorithm A which solves tight group renaming we show how to solve safe-consensus for g processors with 0/1 inputs (See code in Algorithm 1). We assume A is of the form $A(\mathit{processor\text{-}id},\mathit{group\text{-}id})$, receiving the processor id from $1..N$ and initial group id from $1..M$, and returns the new group ID in range $1..l$, and that A cannot be run more than once with the same processor ID. We also assume that invocation by a single processor in isolation always returns the same result, i.e., the only "non-determinism" is due to concurrency, which is true for any deterministic task.

Lemma 2. *There are two values $k1$,$k2$, s.t., $k1 \neq k2$ and a solo-run of $A(k1, k1)$ returns the same value as a solo-run of $A(k2, k2)$*

Proof. The pairs $< 1, 1 >$, $< 2, 2 >$,...,$< l + 1, l + 1 >$ are all valid values to call A with, since $M > l$ and $N > l$. Since A returns values in the range $1..l$, there are at least 2 of the above pairs for which $A()$ will return the same value in a solo-run.

Denote this returned value k. Let $k1$,$k2$ and k be these values for Algorithm A. (Note that these values can be deduced by running $l + 1$ instances of algorithm A, without knowledge of its internal specification, i.e., leaving it as a black-box).

Notice that if more than one processors access the group-renaming concurrently, even if from the same group, then their outputs cannot be deduced (or determined) ahead of time. Their outputs may depend on their interleaving and other parameters such as their ids.

In an attempt to reach consensus we let all g processors run A with group ID $k1$. If $k1$ renamed to k they decide 0, otherwise they decide 1. By definition of tight group renaming $k1$ renames to the same value for all processors, therefore guaranteeing agreement. To achieve the weak-validity requirement of safe-consensus we let all processors first register in memory and take a snapshot. If a processor sees itself alone it runs $A(k1, k1)$, and if it has input 1 it first runs $A(k2, k2)$. This guarantees that if it runs in solo, either $k1$ or $k2$ rename to k according to whether its input is 0 or 1 respectively.

Weak-Renaming: Note that the above construction applies whether the tight group renaming is weak-renaming or strong-renaming (i.e., adaptive).

Notice: The above algorithm can be extended to multivalue safe-consensus with p values, given a g-tight group renaming algorithm which allows at least p groups, s.t. $M > (p - 1)l$, while still using only one instance of algorithm A.

3.1 At Least $g + 1$ Invocations Are Required to Implement Safe-Consensus

In the above construction we showed that tight group renaming with group size g solves binary g-consensus when it may be invoked $g + 1$ times. Taubenfeld

Algorithm 1. g processor 0/1 safe-consensus using a g-tight group renaming box and R/W memory

```
Shared Variables:
    S[1..$g$] : initially NULL
    temp[1..$g$] : $g$ distinct processor ID's which are neither $k1$ nor $k2$
    A(processor-id,group-id) : a $g$-tight group renaming algorithm instance
procedure consensus(proposal)
1:  S[processor-id] = 'ACTIVE'
2:  SS = atomic snapshot of S[1..g]
3:  if for all i $\neq$ processor-id SS[i] = NULL
4:      if proposal = 1 then run A(k2,k2)
5:      value = A(k1,k1)
6:  else
7:      value = A(temp[processor-id],k1)
8:  end if
9:  if value = k then decide 0 else decide 1
end consensus
```

then raised the question whether this is a lower bound, i.e., can a tight group renaming task with group size g which supports at most g processor invocations solve g-safe-consensus (and thus g-consensus?)

We prove that a g-tight group renaming task that allows at most g processor invocations cannot solve g-consensus, by showing how to implement such a task using only $(g-1)$-consensus objects and atomic R/W memory.

Theorem 3. *g-tight group renaming that may be invoked by at most g processors can be implemented using only $(g-1)$-consensus objects and atomic R/W memory.*

Proof. In Algorithm 2 we present the code to solve g-tight group renaming for at most g processors in groups of size g. We utilize the fact that $n = g$, which means that if there is some group with more than $g-1$ processors in it, then all g processors are from the same group and they can therefore decide some default value since there are no other groups to collide with.

We associate a $(g-1)$-consensus object with each of the possible M initial group ID's. We also utilize the result from [3] which shows that tight group renaming for groups of size $g-1$ can be implemented using only $(g-1)$-consensus objects and atomic R/W memory. All processors register their ID and group ID in memory and take a snapshot. If there are g processors in the snapshot and all are from the same group the processor simply decides on a default value 0. Otherwise it uses its group's $(g-1)$-consensus object to decide on its output as follows: If all processors in the snapshot are from its group, it proposes the default value 0, otherwise it accesses a $(g-1)$-tight group renaming task (constructed from $(g-1)$-consensus objects and r/w memory) and proposes the value returned from that task (Which returns ID's in range $1..l$ and does not collide with 0). All processors which decide according to their group's $(g-1)$-consensus object decide the same value, and the only time a processor does not access the $(g-1)$-consensus object is if it sees all g processors in memory and they all have the same group ID. In this case it returns 0 and so will all others, since they all propose 0 to the consensus object associated with their group.

Algorithm 2. g-tight group renaming for g processors using $(g-1)$-consensus objects and R/W memory

```
    Shared Variables:
        cons[1..$M$] : $M$ $(g-1)$-consensus objects, one for each group
        S[1..$N$] : Shared R/W registers, one for each processor, initially NULL
    procedure tight-group-renaming(processor-id,group-id)
 1:     S[processor-id] := group-id
 2:     SS := atomic snapshot of S[1..N]
 3:     count-group := $|\{i|SS[i] =$group-id$\}|$
 4:     count-total := $|\{i|SS[i] \neq NULL\}|$
 5:     if count-group = count-total = $g$ then return 0
 6:     if count-group = count-total
 7:         propose := 0
 8:     else
 9:         propose := $(g-1)$-tight-group-renaming(processor-id,group-id)
10:     end if
11:     return cons[group-id] (propose)
    end consensus
```

4 Safe-Consensus Implements Consensus

We now show that n-safe-consensus is equivalent to regular n-consensus for any n, therefore resulting in g-tight group renaming implementing g-consensus for any g, which complements the result from [3] to prove that g-tight group renaming and g-consensus are equivalent.

Theorem 4. *Safe-consensus is equivalent to consensus.*

We implement n processor consensus given enough n processor safe-consensus tasks as black-boxes. Two algorithms are presented, the first one is somewhat simpler but requires $O(2^n)$ copies of the safe-consensus task to solve for n processors. The second algorithm uses a slight improvement and requires only $O(n^2)$ copies of safe-consensus.

4.1 Consensus Using $O(2^n)$ Safe-Consensus Tasks

For n processors we assume inductively we can solve consensus for $n-1$ processors. We implement consensus as follows (The code is given in Algorithm 3): Processors $1..n-1$ agree on a value P_A recursively using the lower degree version of the task (Line 2) and write the value to shared memory (Line 3). Processors $2..n-1$ then join processor n to recursively agree on a value using another lower degree version of the task, in which processors $2..n-1$ propose P_A and processor n proposes its own input (Line 6), writing the decision, denoted P_B, to shared memory (Line 7).

They now enter an n-processor safe-consensus task (Line 9), proposing their processor-id, in order to decide between P_A and P_B. If it returns n they decide P_B, otherwise they decide P_A.

Algorithm 3. n-consensus using n-safe-consensus, $(n-1)$-consensus tasks and R/W memory

```
Variables:
    processor-id : ID of the running processor, $1..n$
    $P_A$,$P_B$ : MWMR registers, initially NULL
procedure consensus(proposal)
 1:    if processor-id $< n$
 2:        proposal := $(n-1)$-$consensus_A(proposal)$
 3:        $P_A$ := proposal
 4:    end if
 5:    if processor-id $> 1$
 6:        proposal := $(n-1)$-$consensus_B(proposal)$
 7:        $P_B$ := proposal
 8:    end if
 9:    winner := n-safe-consensus(processor-id)
10:    if winner $= n$
11:        decide $P_B$
12:    else
13:        decide $P_A$
14:    end if
end consensus
```

Complexity: $O(2^n)$ as each recursion calls 2 lower degree instances.

4.2 Consensus Using $O(n^2)$ Safe-Consensus Tasks

In Algorithm 4 we implement consensus using only $O(n^2)$ safe-consensus black-boxes. Again, we assume inductively we can solve consensus for $n-1$ processors.

We split the n processors into 2 groups, processors $1..n-1$ on one side, and processor n as a singleton on the other (Denote its input P_B). Processors $1..n-1$ recursively reach consensus among themselves (Line 10), denote this value P_A, and then work together against processor n. The problem here is that with a single safe-consensus task, processors $1..n-1$ can interfere with each other, causing the safe-consensus task to return some arbitrary value, without processor n even being alive. To solve this issue, we use $n-1$ safe-consensus task instances. All n processors run all $n-1$ instances (Lines 4 and 14). Processors $1..n-1$ each start running at a different safe-consensus task. Notice that this guarantees that if processor n is not active, then at least one of the $n-1$ processors will complete a solo-run of a safe-consensus task (The first one to complete running its first instance) and that task will return P_A by definition. On the other hand, if all processors $1..n-1$ are asleep, processor n successfully completes a solo-run of all $n-1$ safe-consensus tasks (Line 4), and they all return P_B. We then have processors $1..n-1$ perform an OR on all their runs, i.e., if at least one of the $n-1$ instances returned P_A they decide on it. Processor n performs an AND, if all instances returned P_B it decides on it. If a processor from $1..n-1$ does not receive P_A from any of the tasks (Line 16), then P_B must be written in memory and it can check if processor n was satisfied (Line 17). If it was not, then they default to deciding P_A. On the other hand if processor n did not receive P_B from all the tasks (Line 6), then P_A must be written in memory and it defaults to P_A (Line 7) as will processors $1..n-1$.

Algorithm 4. n-consensus using $O(n^2)$ safe-consensus tasks and R/W memory

```
Variables:
    $P_A$,$P_B$ : MWMR registers for processors $1..n-1$ and $n$ respectively
    safe-consensus[$1..n-1$] : $n-1$ n-processor or more safe-consensus tasks
    values[1..n-1] : Local registers for each processor
procedure consensus(proposal)
1:      if processor-id = $n$
2:          $P_B$ := proposal
3:          for i := 1 to $n-1$
4:              values[i] := safe-consensus[i](proposal)
5:          end for
6:          if values[i] = proposal for all $i = 1..n-1$ then decide $P_B$
7:          else decide $P_A$
9:      else
10:         proposal := $(n-1)$-consensus (proposal)
12:         $P_A$ := proposal
13:         for i := 1 to $n-1$
14:             values[i] := safe-consensus[$((i + processor_{id}) mod (n-1)) + 1$](proposal)
15:         end for
16:         if values[i] = proposal for some $i = 1..n-1$ then decide $P_A$
17:         elseif values[i] = $P_B$ for all $i = 1..n-1$ then decide $P_B$
18:         else decide $P_A$
20:     end if
end consensus
```

Complexity: $n*(n-1)/2$ as it uses $n-1$ safe-consensus tasks and recursively calls one lower degree instance.

Corollary 5. *g-tight group renaming solves g-consensus using* $n*(n-1)/2$ *such black-boxes, and is therefore a g-consensus task for any g.*

5 Safe Set-Consensus

What about (n, k)-set-consensus that may deliver arbitrary values? There are a few possibilities to generalize safe-consensus to a task (n, k)-safe-set-consensus. It may behave as (n, k)-set-consensus so long as no more than k processors initially access the task simultaneously. In case more than k processors initially access it simultaneously, it is allowed to return invalid results (So long as the k-agreement requirement is still satisfied).

5.1 (n,k)-Safe-Set-Consensus

Theorem 6. *Excluding the pair* $(4, 2)$*,* (n, k)*-set-consensus is implementable from* (n, k)*-safe-set-consensus* iff $n = k + 1$*.*

For $n > k + 1$, we show we cannot solve (n, k)-set-consensus using (n, k)-safe-set-consensus, except for the case of (4,2). We show how to implement (n, k)-safe-set-consensus using a regular $(k, k-1)$-set-consensus task for all n (Code in Algorithm 5): Each processor registers in memory and takes an atomic snapshot. If it sees at most k processors (at most k will), it runs the $(k, k-1)$-set-consensus task, posts the result, and decides it. If a processor sees more than k in its snapshot, then if it sees some posted output it decides it, otherwise it defaults to 0. The only time a processor sees more than k in its snapshot but does not see a posted output is if more than k are concurrently executing the task and no processor has returned yet, so deciding an invalid value 0 in this case is allowed.

Since regular $(k, k-1)$-set-consensus can't solve (n, k)-set-consensus for $n/k > k/(k-1)$ [4], (n, k)-safe-set-consensus can't solve (n, k)-set-consensus for $n > k + 1$, except for the (4,2) case, which is not yet classified.

5.2 Strong (n,k)-Safe-Set-Consensus

Here we strengthen the definition of safe-set-consensus so it can implement regular set-consensus: If an invalid value is returned, *all* processors output that value.

We show that this definition of the task does implement (n, k) set-consensus for all n (See code in Algorithm 6). We use the same idea as in Algorithm 3: We assume inductively we can solve for $(n-1, k)$. Processors $1..n-1$ agree upon k inputs using our $(n-1, k)$ solution and post the output to a shared vector A[] (Line 3). Processors $2..n-1$ each take their results and join another $(n-1, k)$ set-consensus instance with processor n and post the output to a shared result vector B[] (Line 7). Each processor then runs the (n, k) safe-set-consensus task with its processor ID to decide on up to k winners (Line 9). If the winner is n they go to B[] otherwise they go to A[]. If the 'winner' already wrote a value in its location in the vector, that value is chosen, if not it means the result was

Algorithm 5. (n, k) Safe-set-consensus using a $(k, k-1)$ set-consensus task and R/W memory

```
Shared Variables:
    S[1..$n$] : initialy NULL
    RES[1..$n$] : initialy NULL
procedure Safe-set-consensus(proposal)
1:  S[processor-id] := 'ACTIVE'
2:  SS := atomic snapshot of S[1..$n$]
3:  count := $|\{i|SS[i]$ ='ACTIVE'$\}|$
4:  if count $\leq k$
5:      val := $(k, k-1)$-set-cons(proposal)
6:      RES[processor-id] := val
7:      decide val
8:  else
9:      if RES[i] = NULL for all $i = 1..n$ decide 0
10:     else decide any RES[i] s.t. RES[i] $\neq$ NULL
11: end if
end Safe-set-consensus
```

invalid and the processors choose any value posted in that array (By the new agreement definition, they all will go to the same array, A[] or B[]).

6 Conclusions

Few questions are left open. We have shown the connection between g-tight group renaming and g processors safe-consensus. That is, if the members of each group must decide on 1 value (new name), then it is equivalent to consensus between the members of the group. The interesting question is then if the members of a group are allowed to decide on two different values, is that equivalent to some form of 2-(safe)-set-consensus? Or in general to try to find a connection between a variant of g-group-renaming which allows up to k new names per group, and the (g, k)-safe-set-consensus problem. Is this variant of group renaming weaker, stronger or equivalent to set-consensus?

For (n, k)-safe-set-consensus we have shown a strong variant of the problem, which is equivalent to (n, k)-set-consensus, and a weaker variant, which is strictly weaker. Is there, or can there be a tighter characterization of safe-set-consensus in between? What is the power of the weaker variant? In the weaker variant of safe-set-consensus, it still remains to show the classification of the $(4, 2)$-safe-set-consensus. Can it solve $(4, 2)$-set-consensus if it is allowed to return 2 invalid values when 3 or 4 processes access it simultaneously? Another question is whether the Weak-Validity considered in this paper is in some sense the weakest validity condition which is still equivalent to the classical consensus definition.

Algorithm 6. (n, k) set-consensus using Strong (n, k) safe-set-consensus and R/W memory

```
Shared Variables:
    A[1..$n$],B[1..$n$] : Initially NULL
    $(n-1, k)$set-consensus-A/B : 2 $(n-1, k)$ set-consensus tasks
procedure set-consensus(proposal)
 1:     if processor-id $< n$
 2:         proposal := $(n-1, k)$set-consensus-A(proposal)
 3:         A[processor-id] := proposal
 4:     end if
 5:     if processor-id $> 1$
 6:         proposal := $(n-1, k)$set-consensus-B(proposal)
 7:         B[processor-id] := proposal
 8:     end if
 9:     winner := safe-set-consensus(processor-id)
10:     if winner = $n$
11:         if B[n] $\neq$ NULL decide B[n] else decide any B[i] s.t. B[i] $\neq$ NULL
12:     else
13:         if winner in 1..n and A[winner] $\neq$ NULL decide A[winner]
14:         else decide any A[i] s.t. A[i] $\neq$ NULL
15:     end if
end set-consensus
```

Acknowledgements. We are in debt to Gadi Taubenfeld for helpful discussions and suggestions.

References

1. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic Snapshots of Shared Memory. Journal of the ACM 40(4), 873–890 (1993)
2. Attiya, H., Bar-Noy, A., Dolev, D., Peleg, D., Reischuk, R.: Renaming in an asynchronous environment. J. ACM 37(3), 524–548 (1990)
3. Afek, Y., Gamzu, I., Levy, I., Merritt, M., Taubenfeld, G.: Group renaming. In: Baker, T.P., Bui, A., Tixeuil, S. (eds.) OPODIS 2008. LNCS, vol. 5401, pp. 58–72. Springer, Heidelberg (2008)
4. Borowsky, E., Gafni, E.: The Implication of the Borowsky-Gafni Simulation on the Set-Consensus Hierarchy. Technical Report 930021, Department of Computer Science, UCLA (1993)
5. Afek, Y., Gafni, E., Rajsbaum, S., Raynal, M., Travers, C.: Simultaneous consensus tasks: A tighter characterization of set-consensus. In: Chaudhuri, S., Das, S.R., Paul, H.S., Tirthapura, S. (eds.) ICDCN 2006. LNCS, vol. 4308, pp. 331–341. Springer, Heidelberg (2006)
6. De Prisco, R., Malkhi, D., Reiter, M.: On k-Set Consensus Problems in Asynchronous Systems. IEEE Transactions on Parallel and Distributed Systems 12(1), 7–21 (2001)
7. Chaudhuri, S.: More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. Information and Computation 105, 132–158 (1993)

8. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of Distributed Consensus with One Faulty Process. Journal of the ACM 32(2), 374–382 (1985)
9. Gafni, E.: Group-solvability. In: Guerraoui, R. (ed.) DISC 2004. LNCS, vol. 3274, pp. 30–40. Springer, Heidelberg (2004)
10. Guerraoui, R., Kuznetsov, P.: The gap in circumventing the impossibility of consensus. J. Comput. Syst. Sci. 74(5), 823–830 (2008)
11. Herlihy, M.P.: Wait-Free Synchronization. ACM Transactions on Programming Languages and Systems 13(1), 124–149 (1991)
12. Herlihy, M.P., Wing, J.M.: Linearizability: a Correctness Condition for Concurrent Objects. ACM Transactions on Programming Languages and Systems 12(3), 463–492 (1990)

A Appendix

A.1 Algorithm 1 Proof of Correctness

Wait-Free: Since A() is wait-free and there are no loops.

Agreement: By the code all processors decide according to the value returned by $A()$ when invoking it with group ID $k1$. By the definition of tight group renaming this value is the same for all invocations, and therefore all the processors receive either k and decide 0, or a value different than k and decide 1.

Weak-Validity: Let p be a processor that executes the full algorithm in solo. Processor p therefore sees itself alone in its snapshot. If its input is 1 it first runs $A(k2, k2)$ in solo which by construction returns k. Hence by the definition of tight group renaming the call to $A(k1, k1)$ in Line 5 does not return k since $k1 \neq k2$ and this processor decides 1 in Line 9 and the output is valid. If its input is 0 it does not run $A(k2, k2)$ and its solo-run of $A(k1, k1)$ returns k. It therefore decides 0 in Line 9 and the output is valid in this case as well.

It should also be shown that at most g processors invoke A per initial group ID, and that A is never invoked more than once with the same processor ID: For each "real" processor A is run once with group $k1$ and at most one processor runs it with group $k2$, and it is therefore invoked at most g times with $k1$ and at most one time with $k2$ for a total of at most $g + 1$ invocations.

If no processor sees itself alone then all runs of A are with different processor ID's because all the values in $temp[]$ are different. Since at most one processor can see itself alone, at most one will use values $k1$ and/or $k2$ for processor ID's, which are in any case different from all the values in $temp[]$, and therefore $A()$ is never invoked more than once with a given processor ID.

A.2 Algorithm 2 Proof of Correctness

Suppose all processors return in Line 11. In this case each processor decides the result of its group's consensus task. At most $g - 1$ processors from the same group can reach this line, since at most g can access the task, and if they are all from the same group, the last one to take an atomic snapshot in line 2 will attain: $count{-}group = count{-}total = g$ and decide in Line 5. Hence neither the consensus objects nor the $(g-1)$-tight group renaming tasks are ever accessed by more than $g-1$ processors from the same group.

Since the consensus for a certain group always returns the same value, all processors from the same group decide the same value. We need only show that no two groups decide the same value. We claim only one group can see $count-group = count-total$ and propose 0. Since the snapshots are atomic, there cannot be two snapshots in which different group ID's appear alone. Since $(g-1)$-tight-group-renaming(processor-id,group-id) returns a different value for each group by definition, and does not return 0, we have that no two processors from different groups can propose the same value.

We are left with the case that at least one processor does decide in Line 5. This may happen only if all g processors are from the same group and all of them appear in the snapshot. In this case, any processor which sees all g in its snapshot returns 0. All others propose 0 since they all see only their group in Line 6, therefore all proposals are 0 and the consensus in Line 11 must return 0 as well, and all processors return 0.

A.3 Algorithm 3 Proof of Correctness

Wait-Free: By induction and observing the code, since there are no loops.

Agreement: Let v be the value returned at Line 9, which by definition is the same for all processors. Only a single value is written to P_A at Line 3, the value returned by the consensus at line 2, and only a single value is written to P_B at line 7, the result of the consensus at line 6. Therefore we only need to show that if $v = n$, then P_B is not NULL at Line 11, and if $v \neq n$ then P_A is not NULL at line 13.

Suppose a processor finishes executing Line 9, and either P_A or P_B have not yet been written to. If P_B has not yet been written to then this must be processor 1 since processors $2, \ldots, n$ go through line 7. We show that v cannot be n in this case. Since no processor from $2, \ldots, n$ has executed Line 7 then processor 1 runs line 9 in solo and by definition receives 1. Now suppose P_A has not yet been written to and a processor completed line 9. This can only be processor n and processors $1, \ldots, n-1$ have not yet executed Line 3. Therefore processor n runs in solo at Line 9 and by definition receives back n and the decision is therefore P_B.

Validity: We need only to show that the values written to P_A and P_B are valid, i.e., they are proposals of active processors. The result of the consensus at Line 2 is valid, since each processor from $1, \ldots, n-1$ proposes its own input there, therefore P_A is valid. P_B is the result of the consensus at line 6. The proposals there are either P_A, which is valid, or n's input if processor n is alive, thus the result is valid either way.

A.4 Algorithm 4 Proof of Correctness

Wait-Free: By induction: Version n has a constant number of iterations in its loops $(n-1)$, and uses version $n-1$ of our algorithm once.

Lemma 7. *Only a single value is ever written to P_A and a single value to P_B, and any time a processor reads one of these registers in the code, it contains that single value.*

Proof. Processor n only writes once to P_B, at the beginning of its code. Processors $1..n-1$ each write only once to P_A, first thing after they reach agreement with each other at Line 10, therefore only one value is written to P_A, the value agreed upon at Line 10. Now suppose processor n reads P_A, it therefore reached Line 7. It reaches Line 7 only if not all safe-consensus instances return its proposal. Hence processor n did not run a solo-run in at least one of the tasks. Hence at least one of the other processors from $1..n-1$ ran at least some part of that safe-consensus task, and it therefore executed its Line 12, therefore P_A contains its value when processor n reads it. Suppose one of the processors from $1..n-1$ reads P_B. It therefore reached Line 17. It reaches Line 17 only if none of the safe-consensus tasks returned its proposal. Suppose by contradiction that processor n has not executed Line 2 yet. Let p be the first processor from $1..n-1$ to finish running the first safe-consensus task it ran (At least one has finished, since a processor has reached Line 17). Since processor n has not yet executed line 2, it has not yet executed any part of any of the safe-consensus tasks. Since each of the processors $1..n-1$ start with a different task instance, processor p must therefore have completed a solo-run of its first instance. Since all processors $1..n-1$ use the same proposal, and each task guarantees consensus, then all processors $1..n-1$ will decide at Line 16, in contradiction to the fact that a processor from $1..n-1$ reached Line 17. Therefore processor n has executed line 2, and its proposal is in P_B whenever any processor reads P_B.

Validity: Since the values written to P_A and P_B are one of the processor's initial proposals (Either processor n's proposal, or the agreed upon proposal of processors $1..n-1$) and from the fact that the only values decided on in the code are either one of these proposals, or the contents of P_A or P_B, it holds that each processor decides on a valid value.

Agreement: If at least one of the safe-consensus tasks return the agreed upon proposal of processors $1..n-1$, they all decide that value at Line 16. If processor n had that same proposal, then whether it decides at Line 6 or 7, it will be that value (From the previous lemma we have that it will read that proposal form P_A at line 7). If it did not have the same proposal, it will not decide at Line 6, since at least one of the safe-consensus tasks returned a value which is not its proposal. It therefore decides P_A at Line 7, which is the agreed upon proposal of processors $1..n-1$, i.e., all processors decide the same value.

If all safe-consensus tasks return the proposal of processor n, then processor n decides that value at Line 6. If processors $1..n-1$ agreed upon that same proposal, they too all decide that value at line 16 (Since if all tasks returned it, then at least one did as well). If processors $1..n-1$ had a different agreed upon value, then they will not decide at Line 16, since all tasks returned a different value than their proposal. They therefore read that value at Line 17 (As shown in the previous lemma), see that all the tasks returned it, and decide that value there.

Otherwise, at least one safe-consensus task returned a value different from processor n's proposal, and none of them returned the agreed upon proposal of processors $1..n-1$. Processor n therefore will not decide at line 6 and will decide P_A at line 7 (Which is updated at this point). Processors $1..n-1$ will not decide at neither lines 16 nor 17 (Since both those checks will fail), and they too decide P_A, and we reach consensus.

A.5 Algorithm 5 Proof of Correctness

Wait-Free: Since $(k, k-1)$-set-cons is wait-free so is Algorithm 2 as it has no loops.

k-Agreement: The set of all possible values decided on in the algorithm are the $k-1$ values returned by the $(k, k-1)$-set-consensus task and 0, thus at most k values are decided upon, and at most one of them is an invalid value.

Weak-Validity: By the definition of safe-set-consensus, we need to show that if the first processor to output does so after at most $k-1$ others invoked, then all processors decide on a proposed input value. Suppose the first processor to output does so after no more than $k-1$ others have invoked. Denote this processor p. Since at most $k-1$ others have invoked, they all saw at most k processors in their snapshot and accessed the set-consensus object at Line 5 with their proposal. Since this is a standard set-consensus object, it always returns valid values and all these processors therefore decide valid values. Let q be some processor which invoked after p decided and suppose it saw more than k in its snapshot. Since p already decided, it already executed Line 6 and its output is written in memory. Processor q therefore sees a non-NULL value in RES[] and decides it, i.e., no processor decides 0 and all outputs are valid.

A.6 Algorithm 6 Proof of Correctness

Wait-Free: By induction: Since version $n-1$ is wait-free so is our code as we have no loops.

k-Agreement: We split the proof into two, according to whether the safe-set-consensus at Line 9 returned an invalid value or not. If it did not, then by definition it returned at most k valid values, i.e., at most k processor ID's which proposed themselves at Line 9. Hence if the winner is n, B[n] is not NULL (Since n passed Line 7 in order to propose itself), and if the winner is i in $1..n-1$ A[i] is not NULL since it must have passed Line 3 in order to propose itself. Thus for each 'winner' at Line 9 exactly one result is decided on at Line 13 or Line 11. Since there are at most k 'winners', and since each A[] and B[] are written only once we have that at most k values are decided on.

Now suppose Line 9 returns an invalid result i. By definition all processors received this result. If the result is n then all processors decide at Line 11 either B[n] or any non-NULL value in B[], while if the result is not n they decide at lines 13 or 14 either A[i] or any non-NULL value in A[]. Since A[] and B[] are each filled respectively with results of separate k-value set-consensus tasks, then at most k values are chosen. It remains only to show that each processor indeed has some value to choose, i.e., if it needs to decide some non-NULL value in A[] or B[], then there exists such a value at that point: Since Line 9 returned an invalid result, then by definition more than k processors invoked before the first processor returned. Hence at least 2 processors invoked. Therefore any decision occurs after at least 2 processors reached Line 9. These processors are either 1 and n, or at least one of them is from $2..n-1$ and in any case at least one of them wrote to A[] and one of them to B[] before reaching Line 9, therefore we will always have a non-NULL value to choose.

Validity: All proposals to the first k-set-consensus task are valid, since they are inputs of active processors. Since this is a regular set-consensus task all its results are valid, and $A[]$ is filled with valid inputs. Since all proposals to the second k-set-consensus are either results of the first one or the input of processor n, it too returns only valid values, therefore $B[]$ is filled with valid values as well. Since all values decided on are non-NULL values from $A[]$ and $B[]$ it holds that all outputs are valid.