Splitting and Renaming with a Majority of Faulty Processes

David Bonnin
LaBRI, U of Bordeaux, France
bonnin@labri.fr

Corentin Travers
LaBRI, U. of Bordeaux, France
travers@labri.fr

ABSTRACT

Moir and Anderson *splitters* are simple objects, implementable with read/write registers, that return *directions* in {*right*, *down*, *stop*}. Not every process that accesses the object obtains the same direction, and in addition at most one obtains stop. Both in their one-shot and long-lived form, splitters are basic building block of elegant renaming algorithms in shared memory.

In a message passing system when less than half of the processes may fail, splitter can be implemented by first simulating shared registers. This is no longer the case if half of more of the processes may fail. We define and implement one-shot and long-lived splitters suited to the majority of failures environment. Our generalized splitters retain most properties of the original splitters, except that they only guarantee that at most $\lfloor \frac{n}{n-f} \rfloor$ processes return *stop*, where n is the number of processes and $f < n$ an upper bound on the number of failures. We then adapt Moir and Anderson grid of splitters to solve a one-shot and long-lived variant of renaming in which at most $k = \lfloor \frac{n}{n-f} \rfloor$ processes may obtain the same name. One of the main challenge consists in composing long-lived generalized splitters.

1. INTRODUCTION

Context.

We consider a system in which n asynchronous processes communicate by exchanging messages. Although the communication system is supposed to be reliable, processes may fail by crashing. When the number of failures f is bounded by $\lfloor \frac{n}{2} \rfloor$, i.e., a majority of the processes is non-faulty, it is well known that that model is equivalent to the shared memory model [6]. In particular, the simulation in [6] enables any shared memory algorithm to be automatically implemented in message passing. The situation is different when half or more of the processes may fail. Due to asynchrony, the system may suffer from *partitions* for arbitrary long period of time. In this context, finding useful message passing equivalent of basic shared memory building blocks remains challenging. We take up this challenge for two related simple abstractions, namely *splitters* and *renaming*.

Splitter and renaming.

A *splitter* [20, 21] is a shared object that provides an operation called SPLITTER(). This operation returns a *direction* among {*right*, *down*, *stop*}. A splitter has a mutual exclusion flavor [17], as at most one of the invoking process may capture the splitter, that is, gets back *stop* from the object. As it names indicates, a splitter enables partitioning the processes as it guarantees in addition that not all invocations return the same direction *right* or *down*. Splitters have been introduced implicitly by Lamport to implement fast mutual-exclusion in failure-free system [20]. Later, they were captured explicitly as objects by Moir and Anderson [21] to solve renaming [7] in shared memory.

M-Renaming [7] is a fundamental distributed problem, in which participating processes with identities in an arbitrary large set are required to acquire distinct names in a smaller domain, of size M. A simple and elegant shared-memory splitter-based renaming algorithm has been presented by Moir and Anderson [21].

Splitter and renaming can be made *long-lived*. In *long-lived M-renaming*, processes repeatedly acquire and release names in some domain of size M in a such a way that no names are simultaneously acquired by two or more processes. Similarly, a *long-lived splitters* can be invoked repeatedly by the processes. It provides an additional operation, called RELEASE(), which allows a process that has captured the splitter to release it. A each point in time, a long-lived splitter is captured by at most process, and if it not in use (i.e., it is not captured and no operation on the object is pending), it behaves as its one-shot counterpart.

Contributions of the paper.

The paper defines and implements one-shot and long-lived splitter and renaming variants suited for asynchronous message passing system in which a majority of the processes may fail. As partitions may occur, no implementations can guarantee that a splitter is captured by at most one process at a time. Similarly, no renaming implementation can ensure that each new name is acquired by at most one process. The paper makes the following contributions:

1. Definition of *k-splitter* and (M, k)-*renaming* (Section 2). Essentially, a k-splitter behaves like a splitter object, except that at most k processes may simultaneously capture it. Similarly, names acquired by invoking

a (M, k)-renaming object might by shared by at most k processes.

2. Implementation of one-shot and long-lived k-splitter, for $k \geq \lfloor \frac{n}{n-f} \rfloor$ (Sections 3 and 4). Our algorithms generalized Moir and Anderson original shared memory implementations [21]. Special care should be taken to tolerate partitions.

3. Implementation of one-shot (M, k)-renaming, for $k \geq \lfloor \frac{n}{n-f} \rfloor$ (Section 5). For the one-shot case, in any implementations of $(M, 1)$-renaming based on a network of splitters, i.e., [5, 21], k-splitters can be used instead while retaining the same size of the name domain. This way, we obtain implementations of (M, k)-renaming for $M = O(n^2)$ (from Moir and Anderson grid [21]) and $m = O(n^{3/2})$ (from Aspnes smaller splitters network [5]). We also observe that a better name domain of size $2n - 1$ can be obtained by using the seminal message-passing algorithm of Attiya et al. [7].

4. Implementation of long-lived (M, k)-renaming, for $k \geq \lfloor \frac{n}{n-f} \rfloor$ (Section 6.2) and $M = O(n^8)$. Following, e.g. [2, 21], our implementation relies on a grid of long-lived k-splitters. However, composing long-lived k-splitters is not as straightforward as in the one-shot case. Due to partition and asynchrony, different groups of processes may have different perspectives on the state of each splitter object. In fact, the same process might be seen by different processes having a pending operation on several splitters. To cope with these difficulties, we resort to larger grids of splitters and carefully compose the implementation of the splitters in the grid.

Related work.

Renaming is a fundamental problem in distributed computing and a significant body of work has been devoted to study its complexity and solvability, essentially in asynchronous crash-prone shared memory systems,e.g., [3, 2, 1, 7, 8, 10, 11, 12, 13, 19, 21]. In message passing system, renaming implementations have been considered mainly in synchronous systems for various fault models [4, 14, 15, 16, 22, 23]. An asynchronous message-passing algorithm with name domain of size $n+f$ appears in [7]. This algorithm tolerates $f < \frac{n}{2}$ processes failures. In this setting, shared memory renaming algorithms can be turned into message passing algorithms as registers can be simulated when $f < \frac{n}{2}$ [6].

When $f \geq \frac{n}{2}$, partitions may occur and it follows from the CAP theorem [18] that no partition-tolerant asynchronous algorithm can ensure both consistency (here, name-uniqueness) and termination (every non-faulty process trying to acquire a new name eventually succeeds). We relax consistency by allowing new names to be shared by up to $\lfloor \frac{n}{n-f} \rfloor$ processes and, in the long-live case, by also increasing the size of the new name domain. In this regard, the paper might be seen as part of a larger question, namely, how far consistency must be relaxed when up to a given number of partitions have to be tolerated, as in for example geographically distributed systems.

Splitter are basic building blocks in shared memory adaptive renaming algorithms [5, 2, 21]. A renaming algorithm is adaptive if its time/space complexity as well as the size of the domain of new names depends on the number of processes trying to acquire new names. The splitters-based renaming algorithms in this paper are also adaptive in the size of the new name domain.

2. MODEL AND DEFINITIONS

Model of computation.

We assume a standard message-passing model, as described in textbooks, e.g., [9, 24], consisting in n asynchronous processes $\{p_1, \ldots, p_n\}$. Processes communicate by exchanging over a reliable, fully-connected and asynchronous network. This means that each message sent by p_i to p_j is received by p_j after some finite, but unknown, time. Channels are assumed to be *FIFO*, that is for any pair of processes p_i, p_j, the order in which the messages sent by p_i to p_j are received is the same as the order in which they are sent.

The system is equipped with a global clock whose ticks range $\mathbb{T}$ is the positive integers. This clock is not available to the processes, it is used from an external point of view to state and prove properties about executions. An *execution* consists in a (possibly infinite) sequence of *steps*. In each step, a process may send a message to some other processes, performs arbitrary local computation and receives messages. Processes may fail by *crashing*. A process that crashes prematurely halts and never recovers. In an execution, a process is *faulty* if it fails and *correct* otherwise. f denote an upper bound on the maximal number of processes that may fail.

k-Splitters.

As indicated in the introduction, a *one-shot k-splitter* supports one operation called SPLITTTER() that can be invoked at most once by each process. It takes as parameter the identity of the caller. The operation returns a value in the set $\{right, down, stop\}$ subject to the following conditions:

1. Every invocation of SPLITTER(id_i) by a non-faulty process p_i eventually returns.

2. If only one process invokes SPLITTER(id), that process gets back *stop*.

3. If p processes invoke SPLITTER(), at most $p-1$ of them obtain *down* and at most $p-1$ of them obtain *right*.

4. Among the processes that invoke SPLITTER(), at most k of them get *stop*.

We say that a process that obtains a *stop* response has *captured* the splitter.

A *long-lived k-splitter* supports an additional operation, called RELEASE() that allows processes, as its name indicates, to release the splitter after having captured it. As in the one-shot case, an invocation of SPLITTER() returns a direction in $\{right, down, stop\}$. We consider only *well-formed executions* in which (1) each process has at most one pending operation at any point in time and (2) any invocation of RELEASE() is immediately preceded by an invocation by the same process of SPLITTER() and that invocation returns *stop*. At each point of a splitter execution, the splitter is *busy* if

- a process has invoked SPLITTER() and has not yet obtained a response from that invocation or,

- a process has invoked SPLITTER(), obtained *stop* from that invocation and the matching call to RELEASE() has not returned yet.

In addition, the splitter is *idle* at time τ if it is not busy and every message sent by the implementation before time τ has been received. A *busy period* (respectively, *idle period*) is a largest interval in which the splitter is busy (respectively, idle). The splitter is *captured* by process p_i if p_i has invoked SPLITTER() and has obtained a *stop* response from that invocation and has not yet invoked RELEASE(), or that invocation has not yet returned. A long-lived k-splitter has the following property.

1. Every invocation of SPLITTER(id_i) by a non-faulty process p_i eventually returns.
2. At any point in the execution, the splitter is captured by at most k processes.
3. In any busy period immediately preceded by a non-empty idle period, not every invocation of SPLITTER() returns *down* and not every invocation returns *right*.
4. If only one single process invokes SPLITTER() in the execution, each of these invocations returns *stop*.
5. Every period in which every SPLITTER() invocation returns *down* is finite.

Note that property 3 implies that if in a busy period a single process accesses the splitter and this period is preceded by a non-empty idle period, the SPLITTER() invocation in this period returns *stop*.

Renaming.

A *one-shot* (M, k)*-renaming* object is accessed by one operation called GET-NAME(id) that takes as input the identity of the caller. The operation returns a new *name* in the range $[1..M]$ with the following properties:

- (Termination) Any invocation of GET-NAME(id_i) by a non-faulty process p_i returns
- (k-uniqueness) For any name $y \in [1..M]$, at most k invocation of GET-NAME() return y.

A *long-lived* (M, k)*-renaming* object exports an additional operation called RELEASE(). We consider only *well-formed* executions in which each process alternates between invocations of GET-NAME() and RELEASE(), starting with an invocation of GET-NAME(). We say that name $y \in [1..M]$ is *acquired* by process p_i at some point in the execution if p_i has obtained y from a GET-NAME() invocation and the process following invocation of RELEASE() has not returned yet. A long-lived (M, k)-renaming implementation has the same termination requirement as one-shot renaming and must in addition satisfy:

- (Long-lived k-uniqueness) For any name $y \in [1..M]$ and any point in time, name y is acquired by at most k processes.

3. ONE-SHOT SPLITTERS

An algorithm implementing a one-shot $\lfloor \frac{n}{n-f} \rfloor$-splitter appears in Figure 3.1. The idea of the shared-memory implementation of 1-splitter presented in [21] is the following : A process that enters the splitter first writes its name in some shared register, called the *name register*, perhaps overwriting the last name written to this register. Then the process checks if a *door* (represented by a shared Boolean) is open: if not, it returns *right*, and before returning closes the door. If the door is still open, the process checks whether the shared register still contains its own name or not. If not, it returns *down*. Otherwise, it returns *stop*.

The idea of our message-passing algorithm is similar. A process p_i that invokes SPLITTER() first "writes" its name, and then marches through N doors, where $N = O(\frac{n}{n-f})$. If the rth door is found closed, p_i returns *right*. Otherwise, p_i closes the door and checks whether the "name register" still contains its name or not. If a different name is found, p_i returns *down*. Otherwise, the process proceeds to the next door. A process then returns *stop* if it manages to proceed through the N doors, and still have its name in the register.

The name register consists in, at process p_i, a local variable $last_i$. As a majority of the processes may fail, it is not possible to simulate a register with atomic, regular or even safe semantic. Instead, $last_i$ contains the largest process id that p_i has heard of (lines 11 and 17).

Similarly, the local variable $closed_i$ represents the number of doors closed (or, similarly, the rank of the last door closed), as known by process p_i. It is updated each time p_i learns that a door with rank larger than the one it currently knows has been closed (lines 12 and 17). To determine whether the rth door is open or not, process p_i gather the largest closed door known by a quorum of at least $n - f$ processes (lines 6–8). N doors, with $N > \lfloor \frac{n}{n-f} \rfloor$ are necessary to ensure that no more than $\lfloor \frac{n}{n-f} \rfloor$ SPLITTER() invocations return *stop*. Indeed, it may be the case that $f + 1$ processes $p_1, \ldots, p_{f+1}$ see the first door open and be such that $last_i = id_i$. This occurs for example if $p_1, \ldots, p_{f+1}$ checks whether the first door is opened one after the other, in order of their increasing ids and when doing so, gets responses from the same quorum $Q = \{p_{f+1}, \ldots, p_n\}$. Note that each process in quorum Q then learns the largest identity $\max\{id_1, \ldots, id_{f+1}\}$. Thus, in order to pass through the second door, a processes with a lower id must not receive messages from processes in Q (Otherwise, it would learn that a process with higher id has invoked SPLITTER() and return *down*.). So, processes with lower ids should obtain responses from quorum Q' such that $Q' \cap Q = \emptyset$ when checking whether the second door is closed. Intuitively, each new door r allows a new process, whose name is the largest among the processes that have not returned *right* or *down* yet, to lock a new, non-intersecting quorum $Q_r, |Q_r| = n - f$ and $Q_r \cap (\bigcup_{r'<r} Q_{r'}) = \emptyset$ with its identity. Any remaining processes, with lower ids, then returns *down* if it receives a message from any process in a locked quorum while passing through to the next doors $r+1, r+2, \ldots$ As there are at most $\lfloor \frac{n}{n-f} \rfloor$ pairwise disjoint quorums may be formed, it follows that no more than $\lfloor \frac{n}{n-f} \rfloor$ processes may return *stop*.

Proof of the protocol.

Lemma 3.1 proves that every invocation of SPLITTER() by a non-faulty process terminates. The fact that neither *down* nor *right* is the only value returned is proved in Lemma 3.3 and Lemma 3.4 respectively. Finally, Lemma 3.6 shows that at most $\lfloor \frac{n}{n-f} \rfloor$ invocations return *stop*.

Lemma 3.1 (Termination)**.** *For any correct process p_i, the*

Algorithm 3.1 One-shot $\lfloor \frac{n}{n-f} \rfloor$-splitter (code for process p_i)

```
1: initialization
2:     closed_i ← 0; last_i ← −∞;
3: function SPLITTER(id_i)
4:   last_i ← max(last_i, id_i)
5:   for round r_i from 1 to N do        ▷ N = ⌊n/(n−f)⌋ + 1
6:     broadcast(Check, r_i)
7:     wait until n−f messages (AnsCheck, ∗, r_i) have been received
8:     if (∃c ≥ r_i : (AnsCheck, c, r_i) has been received) ∨ (closed_i ≥ r_i) then return right
9:     closed_i ← max(r_i, closed_i); broadcast(Id, last_i, closed_i, r_i)
10:    wait until n − f messages (AnsId,∗, ∗, r_i) have been received
11:    last_i ← max({ℓ : (AnsId, ℓ, ∗, r_i) has been received} ∪ {last_i})
12:    closed_i ← max({c : (AnsId, ∗, c, r_i) has been received} ∪ {closed_i})
13:    if last_i ≠ id_i then return down
14:  return stop;
15: when a message m is received from process p_j :
16:   case m = (Check, r) do send (AnsCheck, closed_i, r) to p_j
17:        m = (Id, ℓ, c, r) do last_i ← max(ℓ, last_i); closed_i ← max(c, closed_i)
18:                          send (AnsId, last_i, closed_i, r) to p_j
```

invocation SPLITTER(id_i) *eventually returns.*

Proof. Let p_i be a correct process. Since at most f of the n processes may fail, and each process, upon receiving a message (Check,r) or (Id,$*, *, r$) from p_i replies with a message (AnsId,$*, *, r$) or (AnsCheck,$*, r$) from p_i, each **wait until** statement (on lines 7 or 10) eventually terminates. As the number of rounds performed by p_i is bounded (by $N = \lfloor \frac{n}{n-f} \rfloor + 1$), it follows that p_i invocation of SPLITTER(id_i) eventually returns. □

By the code, for any process p_i, whenever the variables $closed_i$ or $last_i$ is modified, its value is changed to a larger value (lines 4, 9, 11,12, 17). Hence,

Observation 3.2. *For each process p_i, the successive values of the variables $closed_i$ and $last_i$ form an increasing sequence.*

Lemma 3.3. *Let p be the number of processes that invoke the function* SPLITTER*(). At most $p-1$ processes return* down.

Proof. Let p_m be the process with the largest id that invokes SPLITTER() and let id_m denote its id. At the beginning of the invocation of SPLITTER(), $last_m$ is set to id_m (line 4). Note that the value of the variable $last_m$ can only be increased, as each modification of $last_m$ is of the form $last_m \leftarrow \max(last_m, x)$ where x is an id or a set of ids of some processes that have invoked SPLITTER() (lines 11 and 17). Hence, after p_m has invoked SPLITTER(), we always have $last_m = id_m$. It thus follows that p_m cannot return *down*, since by the code for that to happen, it must be the case that $last_m \neq id_m$ (line 13). □

Lemma 3.4. *Let p be the number of processes that invoke the function* SPLITTER*(). At most $p-1$ processes return* right.

Proof. For each process p_i whose invocation of SPLITTER() returns *right*, let $rr_i, 1 \leq rr_i \leq N$ be the value of r_i when *right* is returned (line 8). Among the processes that return *right*, let p_m be a process whose associated value rr_m is maximal. By the code, in the rr_mth round, p_m receives a message (AnsCheck, $c_m, *$) with $c_m \geq rr_m$ from some process p_i, possibly p_m itself. At process p_i, c_m was the value of $closed_i$ when that message was sent. The value of $closed_i$ is changed to c_m either when p_i performs the c_mth round of the **for** loop without returning *right* (line 9) or when it receives a message (AnsId, $*, c_m, *$) or (Id, $*, c_m, *$) (lines 12 and 17 respectively). In the former case, p_i cannot return *right* as if it does so, this necessarily occurs in an iteration $> c_m$, from which it will follow that $rr_i > rr_m$: a contradiction.

In the latter case, the same reasoning applies to the sender $p_{i1} \notin \{p_m, p_i\}$ of the message (AnsId, $*, c_m, *$) or (Id, $*, c_m, *$). At process p_{i1}, either $closed_{i1}$ is changed to c_m on line 9, in which case p_{i1} cannot return *right* or p_{i1} receives a message (AnsId, $*, c_m, *$) or (Id, $*, c_m, *$) from some process $p_{i2} \notin \{p_m, p_i, p_{i1}\}$. As finitely many processes invoke the SPLITTER(), a process $p_{i\ell}$ at which $closed_{i\ell}$ is changed to c_m on line 9 is eventually found. This process cannot return *right*. □

As an invocation of SPLITTER() by any correct process always terminates (Lemma 3.1) and thus returns *down*, *right* or *stop*, next corollary follows from Lemma 3.3 and Lemma 3.4:

Corollary 3.5. *If only one process invokes* SPLITTER*() and that process returns, it returns* stop.

Lemma 3.6. *At most $\lfloor \frac{n}{n-f} \rfloor$ processes return* stop.

Proof. Let p_i be a process that terminates the rth round (lines 5–13) without returning *down* or *right*, for some $r, 1 \leq r \leq N$. That is, p_i then starts round $r+1$ or, if $r = N$, returns *stop*. Let $M(r,i)$ denote that set of messages of $n-f$ messages (AnsId, $*, *, r$) that have been received by p_i in round r and let $Q(r,i)$ be the set of processes that have sent these messages. In addition, let c_i^r and $last_i^r$ be the value of the variable $closed_i$ and $last_i$, respectively, at the end of round r. Note that $last_i^r = id_i$, since p_i does not return *down* (line 13).

Claim C Let p_i and $p_{i'}$ be two processes with identity id_i and $id_{i'}$ respectively that return stop. *Let $1 < r < r' \leq N$. If $id_i > id_{i'}$, $Q(r,i) \cap Q(r',i') = \emptyset$.*
Proof of the claim. Assume for contradiction that $Q(r,i) \cap Q(r',i') \neq \emptyset$ and let p_k be a process in the intersection. By definition of the sets $Q(r,i)$ and $Q(r',i')$, p_k sends $m = (\text{AnsId}, \ell, c, r)$ and $m' = (\text{AnsId}, \ell', c', r')$ to p_i and $p_{i'}$ respectively. We first observe that $\ell = id_i$ and $\ell' = id_{i'}$.

Message m answers the message $(\text{Id}, \ell_i, c_i, r)$ broadcast by p_i. ℓ_i is the value of $last_i$ when this message is broadcast. Since the successive values of ℓ_i forms an increasing sequence (Observation 3.2), $last_i = id_i$ at the beginning of the p_i

invocation of SPLITTER() (line 4) and $last_i = id_i$ at the end of round r, $\ell_i = id_i$.

When the message $(\mathrm{Id}, \ell_i, *, r)$ from p_i is received by p_k, $last_k$ is changed to ℓ_i if ℓ_i is larger than the current value of $last_k$ (line 17) and m is sent to back. The id ℓ carried by m is the current value of $last_k$. Hence, $\ell \geq \ell_i = id_i$.

Finally, $id_i = last_i^r \geq \ell$ since $m \in M(i,r)$ and, before the end of round r, $last_i$ is changed to the largest id carried by the messages in $M(i,r)$ if this id is larger than the current value of $last_i$. It thus follow that $\ell = id_i$. Similarly, we have $\ell' = id_{i'}$.

Note that ℓ and ℓ' are the values of the local variable $last_k$ when messages m and m' are sent, respectively. Since $id_i > id_{i'}$, if follows from Observation 3.2 that m' is sent before m.

Message $m' = (\mathrm{AnsId}, \ell', \mathrm{c}', \mathrm{r}')$ answers to the message $(\mathrm{Id}, \ell_{i'}, c_{i'}, r')$ broadcast by $p_{i'}$ during round r'. $c_{i'}$ is the value of $closed_{i'}$ when this message is sent. By the code (line 9), $c_{i'} \geq r'$. When $(\mathrm{Id}, \ell_{i'}, c_{i'}, r')$ is received by p_k, $closed_k$ is changed to $c_{i'}$ if $c_{i'}$ is larger than the current value of $closed_k$ (line 17). Since m is sent after m' and the successive values of $closed_k$ form an increasing sequence (Observation 3.2), $closed_k \geq r'$ when $m = (\mathrm{AnsId}, \ell, \mathrm{c}, \mathrm{r})$ is sent to p_i. As c is the value of $closed_k$ when m is sent, $c \geq r'$.

At process p_i, at the end of round r, for any message $(\mathrm{AnsId}, *, cl, r) \in M(i,r)$, the value of $closed_i$ is larger than or equal to cl (line 12). It thus follows that $c_i^r \geq c \geq r' > r$. Since $r < N$ and p_i returns *stop*, p_i performs round $r+1$. As the value of $closed_i$ is non-decreasing, $closed_i \geq r+1$ in round $r+1$, from which we conclude that p_i returns *right*: a contradiction. □

For $r, 1 \leq r \leq N$, let E_r denote the set of processes that terminate round r without returning either *down* or *right*. Note that the set of processes that return *stop* is a subset of E_N. We show that $|E_N| \leq \lfloor \frac{n}{n-f} \rfloor$.

Let $E_N = \{p_1, \ldots, p_m\}$ with $id_1 > \ldots > id_m$ and assume for contradiction that $m \geq \lfloor \frac{n}{n-f} \rfloor + 1 = N$. By Claim C, for all $i, j, 1 \leq i < j \leq N$, $Q(i,i) \cap Q(j,j) = \emptyset$. By definition, for all $i, 1 \leq i \leq N$, $Q(i,i)$ is a set of $n-f$ process. Hence, $|\cup_{1 \leq i \leq N} Q(i,i)| = N * (n-f) > n$: a contradiction since the system consists in n processes.

4. LONG-LIVED SPLITTERS

An implementation of a long-lived $\lfloor \frac{n}{n-f} \rfloor$-splitter appears in Figure 4.1. As in the one-shot case, it is inspired by the long-lived algorithm in [21]. The main difference between the one-shot and long-lived implementations in [21], is that each process "cleans up" before leaving the splitter, i.e., before returning *right* or *down*, or, after returning *stop*, by invoking RELEASE(). To avoid cleaning up too much, the door system is slightly different. When closing a door, a process add its own padlock, and then removes it when cleaning up. In that way, if two processes close a door, but only one leave the splitter, the door will still be closed. In order for a door to re-open, every process that has closed this door must remove its padlock.

Our message-passing implementation shares the same structure with our one-shot k-splitters implementation, namely, each process follows the cycle "check if the door is open – return right or close the door – check the last identity – return down or continue". At each process p_i, the local variable $closed_i$ is replaced by an array $Clsd_i$ as each process now closes each door on its own, adding its personnal padlock. That is, $Clsd_i[id]$ is, to the knowledge of p_i, the latest door that the process with identity id has closed. Since in different invocations of SPLITTER(), the same door can be opened and closed by the same process, $Clsd_i[id]$ is associated with a timestamp stored in $TClsd_i[id]$. When leaving the splitter or releasing it, a process p_j with identity id_j removes its padlocks by increasing its timestamp $TClsd_j[id_j]$ and setting $Clsd_j[id_j]$ to 0 (line 17). Thus, from the point of view of p_i, the identities of the processes that have entered the splitter and have not yet left it or released it are those such that $Clsd_i[id] > 0$. A process can thus determines whether its id is the highest among the ids of the processes in the splitter or not (from its point of view) by examining the pair of array $(Clsd_i, TClsd_i)$ (line 14).

Algorithm 4.1 Long-lived $\lfloor \frac{n}{n-f} \rfloor$-splitter (code for process p_i)

```
1: initialization
2: s_i ← 0; for all id do ⟨Clsd_i[id], TClsd_i[id]⟩ ← ⟨0, 0⟩
3: function SPLITTER(id_i)
4:     s_i ← s_i + 1
5:     for round r_i from 1 to N do        ▷ N = ⌊n/(n−f)⌋ + 2
6:         broadcast(Check, ⟨r_i, s_i⟩)
7:         wait until n − f messages (AnsCheck, ∗, ⟨r_i, s_i⟩) have been received
                                           ▷ including p_i's own message
8:         ⟨Clsd_i, TClsd_i⟩ ← MERGE({⟨C, T⟩ : (AnsCheck, ⟨C, T⟩, ⟨r_i, s_i⟩) has been received})
9:         if ∃id : Clsd_i[id] ≥ r_i then RELEASE( ); return right
10:        ⟨Clsd_i[id_i], TClsd_i[id_i]⟩ ← ⟨r_i, TClsd_i[id_i] + 1⟩
11:        broadcast(Id, ⟨Clsd_i, TClsd_i⟩, ⟨r_i, s_i⟩)
12:        wait until n−f messages (AnsId, ∗, ⟨r_i, s_i⟩) have been received
                                           ▷ including p_i's own message
13:        ⟨Clsd_i, TClsd_i⟩ ← MERGE({⟨C, T⟩ : (AnsId, ⟨C, T⟩, ⟨r_i, s_i⟩) has been received})
14:        if max{id such that Clsd_i[id] > 0) ≠ id_i} then RELEASE( ); return down
15:    return stop
16: function RELEASE( )
17:    ⟨Clsd_i[id], TClsd_i[id]⟩ ← ⟨0, TClsd_i[id] + 1⟩;
18:    broadcast(Release, ⟨Clsd_i, TClsd_i⟩)
19: function MERGE(S)
20:    for all id do ⟨C[id], T[id]⟩ ← max{⟨C'[id], T'[id]⟩ : ⟨C', T'⟩ ∈ S}
           ▷ ⟨c', t'⟩ > ⟨c, t⟩ ⟺ t' > t ∨ (t' = t ∧ c' > c)
21:    return ⟨C, T⟩
22: when a message m is received from process p_j :
23:    case m = (Check, ⟨r, s⟩) do send (AnsCheck, ⟨Clsd_i, TClsd_i⟩, ⟨r, s⟩) to p_j
24:        m = (Id, ⟨C, T⟩, ⟨r, s⟩) do ⟨Clsd_i, TClsd_i⟩ ← MERGE({⟨Clsd_i, TClsd_i⟩, ⟨C, T⟩});
25:                                   send (AnsId, ⟨Clsd_i, TClsd_i⟩, ⟨r, c⟩) to p_j
26:        m = (Release, ⟨C, T⟩) do ⟨Clsd_i, TClsd_i⟩ ← MERGE({⟨Clsd_i, TClsd_i⟩, ⟨C, T⟩});
```

Proof of the protocol.

Termination is established in Lemma 4.1. The property that in a busy period following a non-empty idle period, neither *down* nor *right* is the only value returned by SPLITTER() invocations is shown in Lemma 4.4 and Lemma 4.5. Lemma 4.8 then shows that at any point in time, the splitter is captured by at most $\lfloor \frac{n}{n-f} \rfloor$ processes.

Lemma 4.1 (Termination)**.** *For any correct process p_i, any invocation of* SPLITTER*(id_i) by p_i eventually returns.*

Proof. The proof is similar to the proof of Lemma 3.1. It is left to the reader. □

The values of the variables $\langle Clsd_i[id], TClsd_i[id] \rangle$ are ordered first by the value of the timestamp $TClsd_i[id]$ and then, in case of equality, by $Clsd_i[id]$ (e.g., $(c,t) < (c',t')$ iff $t < t'$ or $(t = t'$ and $c < c')$). From the code (line 8, 10, 13, 24 and 26), each time $\langle Clsd_i[id], TClsd_i[id] \rangle$ is updated, it receives a larger value. Hence,

Observation 4.2. *For each process p_i, and each identity id, the successive values of $\langle Clsd_i[id], TClsd_i[id] \rangle$ form an increasing sequence.*

If the splitter is idle at time τ, then for every process and every id, $\langle Clsd_i[id], TClsd_i[id] \rangle = \langle 0, t \rangle$ where t is the last value assigned to $TClsd[id]$ by the process with identity id. Moreover, there is no pending message. Thus, after an idle period, $Clsd_i[id_j] = 0$ while p_j does not change the value of $Clsd_j[id_j]$. Hence,

Observation 4.3. *After an idle period, $Clsd[id_j] = 0$ until p_j invokes* SPLITTER*().*

Lemma 4.4. *In a busy period that immediately follows a non-empty idle period, not every* SPLITTER*() invocation returns* down*.*

Proof. The lemma is true if at least one invocation never returns (in this case the busy period is infinite.). In the following, we thus assume that every invocation of SPLITTER() made in this busy period returns.

Assume for contradiction that every invocation of SPLITTER() returns *down*.

Consider the set P_I of all processes that invoked SPLITTER() during this period, and let us note p_{max} the process with the largest identity id_{max} in P_I.

For a process p_i with identity id_i to return *down*, it has to verify $max(id$ such that $Clsd_i[id] > 0) \neq id_i$ (line 14). In other words, either $Clsd_i[id_i] = 0$, or $\exists id_j > id_i$ such that $Clsd_i[id_j] > 0$. The first case is impossible, since the $(Clsd_i[id_i], TClsd_i[id_i])$ are in increasing order, and because the last one (overwriting others) is generated by p_i itself in line 10, with $Clsd_i[id_i] > 0$.

Thus, if p_{max} returns *down*, this means that $\exists id_j > id_{max}$ such that $Clsd_{max}[id_j] > 0$. Because just before the busy period, the splitter was idel, this means that, from observation 4.3, process p_j must have invoked SPLITTER() after the idle period and before the moment where p_{max} returns. Thus, process p_j has invoked SPLITTER() during this busy period, and $id_j > id_{max}$. This contradicts the definition of p_{max}.

Hence, not all SPLITTER() invocations may return *down* during this busy period.

□

Lemma 4.5. *In a busy period that immediately follows an non-empty idle period, not every* SPLITTER*() invocation returns* right*.*

Proof. If an invocation of SPLITTER() does not return, the lemma is true. In the following, we assume that in the first busy period, every invocation of SPLITTER() returns.

For an invocation by process p_i to return *right*, it must be the case that $Clsd_i[id] \geq r_i$ (line 9) for some process identity id. Let r_m denote the largest round r such that, for some process p_i and some identity id, there is an invocation of SPLITTER() by p_i during this period such that, when this invocation returns, $Clsd_i[id_i] \geq r$. r_m is well defined, as for each process p_i, $r_i \leq N$.

Let inv be an invocation by some process p_i that returns *right* and, when *right* is returned (line 9), for some id_m, $Clsd_i[id_m] = r_m$. Let p_m be the process whose identity is id_m. This means process p_m changes the value of $Clsd_m[id_m]$ to r_m at line 10 while performing some invocation inv_m of SPLITTER(). By the code, in this invocation, p_m does not return *right* while performing rounds $1, \ldots, r_m$ of inv_m. If *right* is returned in some subsequent round $r > r_m$, then due to the test of line 9, we would have for some id_j, $Clsd_m[id_j] = r_j \geq r > r_m$.

Since this busy period was immediately preceded by an idle period, and because $Clsd[id_j] > 0$ during this busy period, this means by observation 4.3 that p_j has invoked SPLITTER() during this busy period. Moreover, to have $Clsd[id_j] = r_j$, it has to be increased from 0 to r_j in repeated line 10, and thus p_j reached at least $Clsd_j[id_j] = r_j$ before returning. And since $Clsd$ is only increasing, this means that when this invocation returned, $Clsd_j[id_j] \geq r_j > r_m$, thus contradicting the definition of r_m.

Therefore inv_m does not return *right*. □

Lemma 4.6. *Any interval during which every* SPLITTER*() invocation returns* down *is finite.*

Proof. Assume for contradiction that there is an execution and a time τ_1 after which SPLITTER() is invoked infinitely often, and always returns *down*.

Consider the process p_m with the largest identity id_m among the identities of the processes that invoke SPLITTER() infinitely often. For every process p_j with $id_j > id_m$, p_j invokes SPLITTER() a finite number of times. Thus, there is a time τ_2 after which each process p_j with $id_j > id_m$ no longer invokes SPLITTER(), and each such process has either returned from their last invocation or crashed. Between the beginning of the execution and τ_2, there is only a finite number of invocations by processes p_j, $id_j > id_m$, and thus a finite number of different $\langle Clsd[id_j], TClsd[id_j] \rangle$ values.

After time τ_1, every invocation of SPLITTER() by process p_m returns *down*. This can only happen if $Clsd_m[id_j] > 0$ with $id_j > id_m$ at line 14 for some j. But, before checking whether *down* should be returned at line 14, p_m tests if $Clsd_m[id] = 0$ for all id 9. If this is not the case, *right* is returned. Thus, the value of $Clsd_m[id_j]$ changes during the invocation.

We know from observation 4.2 that the values of $\langle Clsd_m[id_j], TClsd_m$ form an increasing sequence. Since there are an infinite number of invocations by p_m that all return *down*, and the number of identity larger than id_m is finite, there is an infinite increasing sequence of $\langle Clsd_m[id_j], TClsd_m[id_j] \rangle$, with $id_j > id_m$ (because there is a finite number of such id_j).

This contradicts the fact that there are only a finite number of $\langle Clsd[id_j], TClsd[id_j]\rangle$ for each $id_j > id_m$. □

Corollary 4.7. *If only a single correct process p_i calls Splitter(), it will necessarily return* stop.

Proof. No other process can have any values that will replace local variables of p_i, because they never calls SPLITTER() and thus never generate new values. Thus, since p_i should call RELEASE() before calling SPLITTER() again, at the beginning of each invocation, $Closed_i[id] = 0$ for any id including id_i. Thus no invocation returns *right*. Since p_i is the only process for which $Clsd[id_i] > 0$, $\max\{id : Clsd[id] > 0\} = id_i$ is always true. Hence no invocation returns *down*. □

Lemma 4.8. *At any point in the execution, the splitter is captured by at most $\lfloor n/(n-f)\rfloor$ processes.*

Proof. Suppose for contradiction that at some time τ, $s > \lfloor n/(n-f)\rfloor$ processes have returned *stop* and have not yet invoked RELEASE(). Let $inv_1, \ldots, inv_s$ denote the last invocation of SPLITTER() preceding τ by those processes. Without loss of generality, assume that p_i is the process that performs invocation inv_i, and let us note its identity id_i.

In round r of invocation inv_i p_i receives a set of $(n-f)$ messages AnsId (line 12). As in the proof of Lemma 3.6, let $Q(r,i)$ denote the set of $(n-f)$ processes that have sent these messages. A key ingredient in the proof is the following claim:

Claim C1. Let $1 < r' < r < N$, and $i, i' \in [1, s]$. If $id_i < id_{i'}$, $Q(r,i) \cap Q(r',i') = \emptyset$.
Proof of Claim C1. Assume for contradiction that $Q(r,i) \cap Q(r',i') \neq \emptyset$ and let $p_x \in Q(r,i) \cap Q(r',i')$.

$p_x \in Q(r,i)$ means that process p_x sent a message M of the form (AnsId,(C_x, TC_x), (r,s)) to p_i (line 25). M is sent as a reply to message (Id,(C_ℓ, TC_ℓ), (r,s)) from p_i, with $C_\ell[id_i] = r$ (line 11). Hence, when M is sent, $C_x[id_i] = r$.

$p_x \in Q(r',i')$ means that process p_x sent a message M' of the form (AnsId,(C'_x, TC'_x), (r',s')) to $p_{i'}$. As above, M' is an answer to a message (Id,(C'_ℓ, TC'_ℓ), (r',s')) sent by $p_{i'}$. Similarily, when M' is sent, $Clsd_x[id_{i'}] = r'$. If M' is sent before M, as the value of $Clsd_x[id_{i'}]$ never decreases until $p_{i'}$ invokes RELEASE(), the value $C_x[id_{i'}]$ carried by M is larger than or equal to $r > 0$ with $id_{i'} \neq id_i$. Thus, after M has been received by p_i, and until $p_{i'}$ invokes RELEASE(), $max(id : Clsd_i[id] > 0) \geq id_{i'} > id_i$ and it follows that inv_i returns *down* (line 14): a contradiction.

If M is sent before M', we have when M' is sent $\langle Clsd_x[id_i], TClsd_x[id_i]\rangle > (r', tc(r'))$ where $tc(r')$ is the value assigned to $TClsd_i[id_i]$ when $Clsd_i[id_i]$ is changed to r' in invocation inv_i. Moreover, at process $p_{i'}$, when M' is received if $Clsd_{i'}[id_i] \leq r'$, then $TClsd_{i'}[id_i] \leq tc(r')$ as p_i has not yet returned from inv_i. Thus, after M' has been received, $Clsd_{i'}[id_i] \geq r > r'$ (line 24). It thus follows that in round $r'+1$ of the invocation $inv_{i'}$, *right* is returned: A contradiction. □

Assume without loss of generality that $id_1 > \ldots > id_s$. Hence, the sets $Q(1,1), Q(2,2), \ldots, Q(N-1, N-1)$ are well defined (since $s \geq N-1$) and pairwise disjoint. Thus $|\bigcup_{1\leq i\leq N-1} Q(i,i)| = (N-1)(n-f) > n$ since $N \geq \lfloor n/(n-f)\rfloor + 2$: a contradiction. □

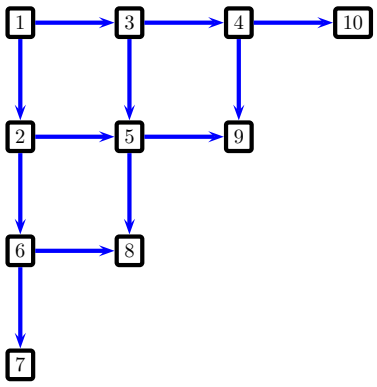

Figure 1: Moir Anderson grid of splitters for 4 processes.

5. ONE-SHOT RENAMING

Ones-shot $(M,1)$-renaming can be implemented with a network of 1-splitters, as presented in [5, 21]. Each splitter in the network is uniquely associated with an integer in $[1..M]$. Starting from a designated splitter, processes traverse the network following the directions obtained by SPLITTER() invocations. Whenever an invocation returns *stop*, the invoking process acquire the corresponding name. Name uniqueness is guaranteed, since for any given splitter, at most invocation returns *stop*. To ensure termination, the network should be large enough. IT must be the case that in any execution, each non-faulty process eventually attains a splitter that no other processes access. Its SPLITTER() invocation on that object then returns stop, and the process thus acquires the name associated with it. The size of the name domain M is determined by the size of the network. For example, in [21], the splitters are organized in half-grid of size $\frac{n(n-1)}{2}$, as pictured in Figure 1. A slightly smaller name domain of $O(n^{3/2})$ can be achieved by organizing the splitters in a different way [5]. If only k-splitters are available, one can implement k-renaming from any algorithm $\mathcal{A}$ for $(M,1)$-renaming based on a splitter network, by simply replacing each 1-splitter in the network by a k-splitter. The implementation retains the same size of the name domain as the original one. Hence, in this way, we obtain message passing implementations of $(\frac{n(n-1)}{2}, \lfloor\frac{n}{n-f}\rfloor)$-renaming from [21] and $(O(n^{3/2}), \lfloor\frac{n}{n-f}\rfloor)$-renaming from [5].

A smaller name domain is achieved by algorithm in [7]. This algorithm implements $(n+f,1)$-renaming in the asynchronous message-passing model and tolerates up to $f < \frac{n}{2}$ failures. A careful examination of its proof reveals that the same algorithm actually implements $(n+f, \lfloor\frac{n}{n-f}\rfloor)$-renaming, for any $f < n$.

6. LONG-LIVED RENAMING

Generalizing the ideas of the message-passing algorithm of [7] to the long-lived case while tolerating $f \geq \frac{n}{2}$ failures results in a non-blocking algorithm. That is, some non-faulty processes may never be able to acquire names. Instead, following [21], our long-lived (M,k)-renaming implementation relies on a network of long-lived k-splitters.

As in a one-shot case, long-lived $(M,1)$-renaming can be implemented by a network of M long-lived 1-splitters. A

process acquires a name by traversing the network until it accesses a splitter from which it gets back *stop*. To release the name, the process invoke RELEASE() on the splitter associated with that name. The size and the structure of the network has to be chosen in such way that in any execution, whatever the names that have been acquired, any process traversing the network eventually reaches a splitter from which it gets back *stop*.

As in every execution of long-lived k-splitter, the splitter is captured by at most k processes at the same time, a network of long-lived k-splitters ensures that no names is acquired simultaneously by more than k processes. For termination, a process accessing a long-lived k-splitter uncontended is ensured to get back *stop* only if the splitter is idle when p_i invokes SPLITTER() (Property 3 of the definition).

The network our long-lived (M, k)-renaming implementation relies on is an half-grid as in [21] (Figure 1). The ith k-splitter in the network is implemented by an instance $\mathcal{S}^i$ of some long-lived k-splitter algorithm $\mathcal{S}$, e.g., the one described in Section 4. We remark however that if the instances are treated as black-box, some processes might not be able to acquire names whatever the size of the grid. To see why, consider an execution in which process p_i enters the k-splitter i_ℓ, after accessing k-splitters $i_1, \ldots, i_{\ell-1}$ by SPLITTER() operations. Due to partitions, groups of processes of size $\geq n - f$ might not be aware that each invocation of SPLITTER() on objects $i_1, \ldots, i_{\ell-1}$ by p_i has returned. Hence, each k-splitters $i_i, \ldots, i_{\ell-1}$ are not idle and, and hence another process p_j may obtain the same directions as p_i when accessing splitters $i_i, \ldots, i_{\ell-1}$. p_j may thus invoke also SPLITTER() on the ℓth splitter, from which it follows that another splitter $i_{\ell+1}$ has to be accessed by p_i before it acquires a name. The argument can be repeated to extend to arbitrary lengths the sequence of splitters accessed by p_i before acquiring a name, while keeping the number of partitions and the number of processes trying to acquire names bounded. The concurrent composition of the instances presented next avoids this issue essentially by bounding the number of pending messages in each channel.

6.1 Concurrent composition of M long-lived k-splitters

Without loss of generality, we assume that algorithm $\mathcal{S}$ implementing a long-lived k-splitter is *full information* and follows a query-response communication pattern. More precisely, each time a message is sent by process p_i , it includes the complete state σ_i of p_i. When such a message is received by some process p_j, p_j changes its states σ_i by calling a function $update(\sigma_j, \sigma_i)$. Moreover, there are two types of messages, namely Query and Answer. Answer are sent back when a Query message is received. Query messages are broadcast, and, after a Query has been broadcast, the sender waits until it receives $(n - f)$ matching Answer messages. For example, in Algorithm 4.1, *Check*, *Id*, and *Release* are Query messages, while *AnsCheck* and *AnsId* are Answer messages.

Algorithm $\mathcal{S}$ consists in n local algorithms, $\mathcal{S}_1, \ldots, \mathcal{S}_n$ one per process. As we want to implement a grid of M splitters, each process p_i executes M instances $\mathcal{S}_i^1, \ldots, \mathcal{S}_i^M$. At each process p_i, instead of executing M independent threads, one per instance, process p_i take step in each instance $\mathcal{S}_i^\ell$ sequentially, one instance after the other in round robin fashion (line 5). In particular, at any time, p_i is waiting for $n - f$ Answer message matching a Query in at most one instance (lines 6–9). Query messages may be *accepted* (line 13) or *rejected* (line 12).

Algorithm 6.1 Composing M long-lived k-splitters (code for process p_i)

```
1: init
2:     for each ℓ ∈ [1..n] do σ_i[ℓ] ← initial state of S_i^ℓ;
       Ans[1, 2, ...] ← [∅, ..., ∅]
3:     s_i ← 0; last_send_i[1..n] ← [0, ..., 0];
       last_rcv_i[1..n] ← [0, ..., 0]; ℓ ← 1
4: while true do
5:     ℓ ← (ℓ mod M) + 1; take a step of S_i^ℓ;
6:     if step of S_i^ℓ is broadcast Query then
7:         lastq_i ← (σ_i, ℓ); s_i ← s_i + 1
8:         for each j ∈ [1..n] do send (Query, (σ_i, ℓ) ,
           last_rcv_i[j], s_i)
9:         Ans[s_i] ← ∅; wait until |rec_i[s_i]| ≥ n − f
10: when a message m is received from process p_j
11:    case m = (Query, (σ, ℓ), lr, s) do
12:        if lr < last_send_i[j] then
           (Reject,last_send_i[j],s) to p_j          ▷ m is rejected
13:        else last_send_i[j] ← last_send_i[j] + 1;    ▷ m is
           accepted
14:            for each ℓ ∈ [1, M] do update(σ_i[ℓ], σ[ℓ]);
           send (Answer, σ_i, last_send_i[j], s) to p_j
15:    case m = (Answer, σ, ls, s) do
16:        last_rcv_i[j] ← ls; Ans[s] ← Ans[s] ∪ {m}; for
           each ℓ ∈ [1, M] do update(σ_i[ℓ], σ[ℓ]);
17:    case m = (Reject, ls, s) do
18:        last_rcv_i[j] ← max{ls, last_rcv_i[j]}; if s = s_i
           then send (Query, lastq_i, last_rcv_i[j], s_i)
```

When at process p_j a Query is accepted, a matching Answer is sent (line 14) and the state of the corresponding instance updated. A query from p_i is accepted by p_j if and only if the last Answer sent by p_j to p_i has been received by p_i before the query is sent. To that end, a counter $last\_send_j[i]$ identifies the Answer sent by p_j to p_i. At process p_i, $last\_rcv_i[j]$ keeps track of the number of the last Answer received from p_j. That value is sent to p_j with each Query message, so that p_j can check whether its last Answer message has been received or not. This mechanism ensures that at any time, at most one of the Queries sent by p_i and not yet received by p_j can be accepted by p_j. Also, because the channels are FIFO and since an Answer message is only sent immediately after a Query message is accepted, there is at most one pending Answer message from p_i to p_j at any time.

Query messages from p_i that are not accepted by p_j (line 12) are discarded, i.e., they are ignored by p_j. To prevent stalling, that is, to prevent p_i from waiting forever to receive $n-f$ matching Answers to one of its Query, p_j sends a Reject message (line 12) asking the query to be sent again. When the Reject message is received by p_i, every Answer from p_j to p_i has been received. Hence, if p_i is still waiting for $n - f$ Answers, the Query that it sends again carries the number of last Answer from p_j and is necessarily accepted by p_j.

Any process p_k traversing the grid of splitters invoke SPLITTER() on one object at a time. Each Query or Answer message sent by p_i carries not only the state $\sigma_i[\ell]$ of the instance $\mathcal{S}_i^\ell$ on behalf of which the message is sent but also

the states $\sigma_i[1], \ldots, \sigma_i[M]$ of every of other instances. When m is received by some process p_j, p_j updates, based on $\sigma_i[1], \ldots, \sigma_i[M]$ the state of each instance $\mathcal{S}_j^1, \ldots, \mathcal{S}_j^M$. Besides ensuring progress in each instance at each non-faulty process despite the fact that some Query messages might be discarded, this also guarantees that at most one of the states $\sigma_i[1], \ldots, \sigma_i[M]$ is a state in which process p_k has an active operation. Hence, at any point in time, in the collection of states σ_i, at most n splitters are active.

Moreover, we know that, at any time, each one of the $O(n^2)$ directed channels contain at most 2 messages m, m' (a Query and an Answer) that will be accepted. In states σ and σ' contained in these messages, at most n splitters are active. Therefore, at any point in time, at most $O(n^3)$ different splitters are seen as active by the processes. The remaining splitters are idle:

Lemma 6.1. *There is a bound $B = O(n^3)$ such that, at any time, at most B splitters are not idle.*

6.2 Long-lived renaming

Finally, we show that for M large enough, a grid of long-lived $\lfloor \frac{n}{n-f} \rfloor$-splitters implements long-lived $(M, \lfloor \frac{n}{n-f} \rfloor)$-renaming, provided that the implementation of the splitter are composed as explained in the previous section.

A *grid* of *depth* D consists in $\frac{D(D+1)}{2}$ splitters denoted $s_{i,j}$ where $1 \leq i, j \leq D$ and $i+j \leq D+1$. For each i, j, $2 \leq i+j \leq D$, splitter $s_{i,j}$ has a *right* arrow pointing towards splitter $s_{i+1,j}$ and *down* arrow pointing towards splitter $s_{i+1,j+1}$. For $1 \leq d \leq D$, *diagonal* d consists in the splitters $\{s_{i,j} : i + j = d + 1\}$.

Each splitter is associated with a unique integer in the interval $[1, \frac{D(D+1)}{2}]$. A process p_i acquiring a name enters the network by invoking SPLITTER() on $s_{1,1}$. It then traverses the grid, following the directions returned by its SPLITTER() invocation, until it gets back *stop*. The name acquired by p_i is the one associated with the splitter from which it gets back *stop*. To release the name it has previously acquired, p_i invokes RELEASE() on the corresponding splitter.

In an execution, we say that a process is *in the diagonal d at time τ* is it has invoked SPLITTER() on a splitter of $s \in d$, and that invocation has not returned by time τ. By extension, for $d \leq d'$, a process is *present in the diagonals $[d, d']$ at time τ* if it is in a diagonal d'' at time τ, for some $d'' \in [d, d']$.

Recall that, by Lemma 6.1, at any time in the execution there is a bound B on the number of non-idle splitters.

Lemma 6.2. *Let $p, 1 \leq p \leq n$ and $d_1 < d_2$ such that $d_2 - d_1 + 1 \geq B + p + 1$. Suppose that at any time, at most p processes p processes are present in the diagonals $[d_1, d_2]$. Then, for any $d_3 \geq d_2$ and any time, at most $p-1$ processes are present in the diagonals $[d_2, d_3]$.*

Proof. Assume for contradiction that there exists an execution during which, at some time τ_f, all p processes are in diagonal d_2 or beyond. Note that to reach a diagonal $d \geq d_2$, process p_i, has successively invoked SPLITTER() on splitters in diagonal $1, 2, \ldots, d-1$ in that order. Let τ_0 be the last time before τ_f when a process was in diagonal d_1.

A splitter is *occupied* at some time τ if an operation has been invoked on that splitter before time τ and this operation has not returned by time τ. Let $S \subset \cup_{d_1 \leq d \leq d} d$ denote the set of splitters that are occupied at some time between τ_0 and τ_f. Let n_d the number of splitters in diagonal d that are also in S, i.e., $n_d = S \cap d$.

We prove that, for any , $d_1 \leq d \leq d_2$, $n_{d+1} - n_d \geq 1 - NI_d$, where NI_d is the number of splitters in diagonal d that are non-idle at time τ_0.

If a splitter s in S and in diagonal d ($d_1 \leq d \leq d_2$) is idle at time τ_0, then it cannot enter a non-idle period unless at least one process accesses this splitter. This means that, either a process accessing it returns *stop*, or at least 2 processes access it during the period $[\tau_0, \tau_f]$. Since every process reach at least diagonal d_2, no process gets back *stop* during the interval $[\tau_0, \tau_f]$, (Otherwise, such a process returns to the initial diagonal in the interval, contradicting the definition of τ_0). Therefore at least 2 processes concurrently access s.

Moreover, as s is idle when the first access starts, because no process may *stop* , both *right* and *down* are returned by s at some point during $[\tau_0, \tau_f]$ (from property 3). Consequently, the two splitters following s in the next diagonal are both in S.

If a splitter s in S and in diagonal d is non-idle at τ_0, then every SPLITTER() invocation on s in the interval $[\tau_0, \tau_f]$ may return the same direction, *right* or *down* (but not *stop*). Thus, at least one splitter following s in the next diagonal is in S.

Then, by an induction on NI_i, we prove that $n_{i+1} - n_i \geq 1 - NI_i$. . Summing these inequalities leads to $n_{d_2} - n_{d_1} \geq d_2 - d_1 - NI_{All}$, where NI_{All} is the number of splitters in the diagonals $[d_1, d_2]$ that are non-idle at time τ_0. As, by Lemma 6.1, $NI_{All} \leq B$, we have $n_{d_2} \geq n_{d_1} + d_2 - d_1 - B \geq n_{d_1} + B + p - B \geq p + 1$, since at least one splitter in diagonal d_1 is occupied at τ_0 (and thus $n_1 \geq 1$).

But, during the interval $[\tau_0, t_f]$, no more than p processes are present in the diagonals $[d_1, d_2]$ of the grid, and each process present in these diagonals keeps progressing from one diagonal to the next. As a process cannot enter more than one splitter in each diagonal, at most p splitters in the diagonal d_2 are in S, i.e. $n_{d_2} \leq p$. This is a contradiction. □

Lemma 6.3. *Let $p \in \mathbb{N}$ and let $D_k = B + k + 1$, for any $k \geq 1$. Let G be a grid of depth at least $D = B + D' = B + 1 + \sum_{k=1}^{p}(D_k - 1)$. If at most p processes use the grid, each process returns stop before reaching the end of the grid.*

Proof. From Lemma 6.2, we know that at any time in the execution, at most $p-1$ processes can be in the part consisting in the diagonal $d \geq D_p$. By applying again Lemma 6.2 to diagonals $d \geq D_p$, it follows that at most $p-2$ processes can be in the diagonals $d \geq D_p - 1 + D_{p-1}$.

Therefore, by induction, we have that at most 1 process can be in the diagonals $d \geq D'$. At the time τ process p_i enters the diagonal D', at most B splitters in the diagonals $[D', D]$ are non-idle (Lemma 6.1). Moreover, no splitter in these diagonals that is idle period at time τ becomes non-idle unless the process p_i enters it and returns *stop*. Thus, p_i can visit at most $B+1$ different splitters before returning *stop*. Since there are $B+1$ diagonals in the diagonal $[D', D]$, then p_i necessarily returns *stop*. □

Hence, the previous lemma implies that the grid G implement long-lived $(M, \lfloor \frac{n}{n-f} \rfloor)$-renaming for $M = O(n^8)$.

Theorem 6.4. *There is a message-passing $(M, \frac{n}{n-f})$-renaming algorithm for $M = O(n^8)$.*

7. CONCLUSION

The paper has investigated partition-tolerant implementations of splitter and renaming. In asynchronous message-passing systems in which $f \geq \frac{n}{2}$ processes may fail, at most $\lfloor \frac{n}{n-f} \rfloor$ partitions may occur. It is thus not possible to guarantee that *stop* is returned to less than $\lfloor \frac{n}{n-f} \rfloor$ from a splitter or that new names in renaming are shared by less than $\lfloor \frac{n}{n-f} \rfloor$ processes. The paper has provided implementations of one-shot and long-lived f-tolerant implementation of $\lfloor \frac{n}{n-f} \rfloor$-splitters. It has also shown that, despite their weak semantic, when appropriately composed, long-lived $\lfloor \frac{n}{n-f} \rfloor$-splitters can be used as a basic building block to implement $(M, \lfloor \frac{n}{n-f} \rfloor)$-renaming, where the size of the name domain $M = O(n^8)$. Obvious direction for future research is to improve the size of the domain of new names.

8. REFERENCES

[1] Y. Afek and M. Merritt. Fast, wait-free (2k-1)-renaming. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '99, pages 105–112, New York, NY, USA, 1999. ACM.

[2] Y. Afek, G. Stupp, and D. Touitou. Long lived adaptive splitter and applications. *Distributed Computing*, 15(2):67–86, 2002.

[3] D. Alistarh, J. Aspnes, K. Censor-Hillel, S. Gilbert, and R. Guerraoui. Tight bounds for asynchronous renaming. *J. ACM*, 61(3):18, 2014.

[4] D. Alistarh, H. Attiya, R. Guerraoui, and C. Travers. Early deciding synchronous renaming in o(logf) rounds or less. In *19th International Colloquium on Structural Information and Communication Complexity (Sirocco)*, volume 7355 of *Lecture Notes in Computer Science*, pages 195–206. Springer, 2012.

[5] J. Aspnes. Slightly smaller splitter networks. *CoRR*, abs/1011.3170, 2010.

[6] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995.

[7] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3):524–548, July 1990.

[8] H. Attiya and A. Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM Journal on Computing*, 31(2):642–664, 2001.

[9] H. Attiya and J. Welch. *Distributed Computing*. Wiley, 2004.

[10] E. Borowsky and E. Gafni. Immediate atomic snapshots and fast renaming. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing*, PODC '93, pages 41–51, New York, NY, USA, 1993. ACM.

[11] J. E. Burns and G. L. Peterson. The ambiguity of choosing. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 145–157. ACM, 1989.

[12] A. Castañeda and S. Rajsbaum. New combinatorial topology bounds for renaming: the lower bound. *Distributed Computing*, 22(5-6):287–301, 2010.

[13] A. Castañeda and S. Rajsbaum. New combinatorial topology bounds for renaming: The upper bound. *J. ACM*, 59(1):3, 2012.

[14] S. Chaudhuri, M. Herlihy, and M. R. Tuttle. Wait-free implementations in message-passing systems. *Theor. Comput. Sci.*, 220(1):211–245, 1999.

[15] O. Denysyuk and L. Rodrigues. Byzantine renaming in synchronous systems with t<n. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 210–219. ACM, 2013.

[16] O. Denysyuk and L. Rodrigues. Order-preserving renaming in synchronous systems with byzantine faults. In *33rd IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2013.

[17] E. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 26(1):21–22, 1983.

[18] S. Gilbert and N. A. Lynch. Perspectives on the CAP theorem. *IEEE Computer*, 45(2):30–36, 2012.

[19] M. Herlihy and N. Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, 1999.

[20] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, Jan. 1987.

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

[22] M. Okun. Strong order-preserving renaming in the synchronous message passing model. *Theor. Comput. Sci.*, 411(40-42):3787–3794, 2010.

[23] M. Okun, A. Barak, and E. Gafni. Renaming in synchronous message passing systems with byzantine failures. *Distributed Computing*, 20(6):403–413, 2008.

[24] M. Raynal. *Communication and Agreement Abstractions for Fault-Tolerant Distributed Systems*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.