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 or 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 \(\left\lfloor \frac{n - f}{2} \right\rfloor\) 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 \(k = \left\lfloor \frac{n - f}{2} \right\rfloor\) 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 \(\left\lfloor \frac{n}{2} \right\rfloor\), 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 \texttt{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 \texttt{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 \texttt{right} or \texttt{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 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 \texttt{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 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 be shared by at most 
k\) processes.

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

3. Implementation of one-shot \((M, k)\)-renaming, for \(k \geq \lceil \frac{n}{2} \rceil \) (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 net-
work [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 \(k \geq \lceil \frac{n}{2} \rceil \) (Section 6.2) and \(M = O(n^3)\). 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 pro-
cesses 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 com-
puting and a significant body of work has been devoted
to study its complexity and solvability, essentially in asyn-
chronous 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 tol-
erates \(f < \frac{n}{2}\) processes failures. In this setting, shared mem-
ory renaming algorithms can be turned into message passing
algorithms as registers can be simulated when \(f < \frac{n}{2}\).

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 \(\lceil \frac{n}{2} \rceil \) 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
be tolerated, as in for example geographically distributed
systems.

Splitter are basic building blocks in shared memory adap-
tive 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 pro-
cesses \(\{p_1, \ldots, p_n\}\). 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 \(T\) 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 execu-
tion consists in a (possibly infinite) sequence of steps. In
each step, a process may send a message to some other pro-
cesses, performs arbitrary local computation and receives
messages. Processes may fail by \(crashing\). A process that
crashes prematurely halts and never recovers. In an execu-
tion, 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 sup-
ports one operation called \(SPLITTER()\) 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 \(\{right, \down, \text{stop}\}\) subject to the following condi-
tions:

1. Every invocation of \(SPLITTER(\text{id})\) by a non-faulty pro-
cess \(p\) eventually returns.
2. If only one process invokes \(SPLITTER(\text{id})\), that process
gets back \(\text{stop}\).
3. If \(p\) processes invoke \(SPLITTER()\), at most \(p - 1\) of them
obtain \(\text{right}\) and at most \(p - 1\) of them obtain \(\text{stop}\).
4. Among the processes that invoke \(SPLITTER()\), at most
\(k\) of them get \(\text{stop}\).

We say that a process that obtains a \(\text{stop}\) response has cap-
tured the splitter.

A \(long-lived\ \(k\)-splitter supports an additional operation,
called \(RELEASE()\) that allows processes, as its name indi-
cates, to release the splitter after having captured it. As in
the one-shot case, an invocation of \(SPLITTER()\) returns a di-
rection in \(\{right, \down, \text{stop}\}\). We consider only \(well-formed\ execu-
tions in which (1) each process has at most one pend-
ing 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
\(\text{stop}\). At each point of a splitter execution, the splitter is
\(busy\) if

- a process has invoked \(SPLITTER()\) and has not yet ob-
tained a response from that invocation or,
a process has invoked \textsc{splitter()}, obtained \textit{stop} from that invocation and the matching call to \textsc{release()} has not returned yet.

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

1. Every invocation of \textsc{splitter}(id) 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 \textsc{splitter()} returns \textit{down} and not every invocation returns \textit{right}.
4. If only one single process invokes \textsc{splitter()} in the execution, each of these invocations returns \textit{stop}.
5. Every period in which every \textsc{splitter()} invocation returns \textit{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 \textsc{splitter()} invocation in this period returns \textit{stop}.

\textbf{Renaming.}

A \textit{one-shot $(M,k)$-renaming} object is accessed by one operation called \textsc{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 \textsc{get-name}(id) by a non-faulty process $p_i$ returns.
- (k-uniqueness) For any name $y \in [1..M]$, at most $k$ invocations of \textsc{get-name()} return $y$.

A \textit{long-lived $(M,k)$-renaming} object exports an additional operation called \textsc{release}(). We consider only \textit{well-formed} executions in which each process alternates between invocations of \textsc{get-name()} and \textsc{release}(), starting with an invocation of \textsc{get-name}(). We say that name $y \in [1..M]$ is \textit{acquired} by process $p_i$ at some point in the execution if $p_i$ has obtained $y$ from a \textsc{get-name}() invocation and the process following invocation of \textsc{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. \textbf{ONE-SHOT SPLITTERS}

An algorithm implementing a one-shot $\lfloor \frac{m}{n} \rfloor$-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 \textit{name register}, perhaps overwriting the last name written to this register. Then the process checks if a \textit{door} (represented by a shared Boolean) is open: if not, it returns \textit{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 \textit{down}. Otherwise, it returns \textit{stop}.

The idea of our message-passing algorithm is similar. A process $p_i$ that invokes \textsc{splitter()} first “writes” its name, and then marches through $N$ doors, where $N = O(\frac{m}{n})$. If the $r$th door is found closed, $p_i$ returns \textit{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 \textit{down}. Otherwise, the process proceeds to the next door. A process then returns \textit{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 \textit{last}. As a majority of the processes may fail, it is not possible to simulate a register with atomic, regular or even safe semantic. Instead, \textit{last} contains the largest process id that $p_i$ has heard of (lines 11 and 17).

Similarly, the local variable \textit{closed}, represents the number of doors closed (or, similarly, the rank of the last door closed), 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 $r$th door is open or not, process $p_i$ gathers the largest closed door known by a quorum of at least $n - f$ processes (lines 6–8). $N$ doors, with $N > \frac{m}{n}$ are necessary to ensure that no more than $\frac{m}{n}$ \textsc{splitter()} invocations return \textit{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 \textit{last}, = \textit{id},. 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$(\textit{id}_1, \ldots, \textit{id}_{f+1})$. 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 \textsc{splitter()} and return \textit{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 \textit{right} or \textit{down} yet, to lock a new, non-intersecting quorum $Q_r = \{Q_{r-1}\} = n - f$ and $Q_r \cap (\bigcup_{i=1}^{r} Q_i) = \emptyset$ with its identity. Any remaining processes, with lower ids, then returns \textit{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 $\frac{m}{n}$ pairwise disjoint quorums may be formed, it follows that no more than $\frac{m}{n}$ processes may return \textit{stop}.

\textbf{Proof of the protocol.}

Lemma 3.1 proves that every invocation of \textsc{splitter()} by a non-faulty process terminates. The fact that neither \textit{down} nor \textit{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 $\frac{m}{n}$ invocations return \textit{stop}.

\textbf{Lemma 3.1 (Termination).} For any correct process $p_i$, the
Algorithm 3.1 One-shot \( \lfloor \frac{n}{\pi - 1} \rfloor \)-splitter (code for process \( p_i \))

1: initialization
2: \( \text{closed}, \leftarrow 0; \ last_i, \leftarrow -\infty; \)
3: function \( \text{SPLITTER}(id_i) \)
4: \( \text{last}_i, \leftarrow \max(\text{last}_i, id_i) \)
5: for round \( r_i \) from 1 to \( N \) do \( N = \lfloor \frac{n}{\pi - 1} \rfloor + 1 \)
6: broadcast(\( \text{Check}, r_i \))
7: wait until \( n - f \) messages (\( \text{AnsCheck}, * \), \( r_i \)) have been received
8: if \( \sum_{3c} \geq r_i \) \( (\text{AnsCheck}, c, r_i) \) has been received \( \lor (\text{closed}_i, \geq r_i) \) then return \( \text{right} \)
9: \( \text{closed}_i, \leftarrow \max(r_i, \text{closed}_i); \) broadcast(\( \text{Id}, \text{last}_i, \text{closed}_i, r_i) \)
10: wait until \( n - f \) messages (\( \text{AnsId}, *, * \), \( r_i \)) have been received
11: \( \text{last}_i, \leftarrow \max(\ell \), \( \text{last}_i) \)
12: \( (\text{AnsId}, \ell, * \), \( r_i \)) has been received \( \cup \{ \text{last}_i \} \)
13: \( \text{closed}_i, \leftarrow \max(c \), \( \text{closed}_i) \)
14: \( (\text{AnsId}, \ast \), \( c \), \( r_i \)) has been received \( \cup \{ \text{closed}_i \} \)
15: if \( \text{last}_i \neq \text{id}_i \) then return \( \text{down} \)
16: return \( \text{stop} \)
17: when a message \( m \) is received from process \( p_j \)
18: case \( m = (\text{Check}, r) \) do send \( \text{AnsCheck}, \text{closed}_i, r \)
19: \( p_j \)
20: \( m = (\ell, c, r) \) do last \( _i, \leftarrow \max(\ell, \text{last}_i) \)
21: \( \text{closed}_i, \leftarrow \max(c \), \( \text{closed}_i) \)
22: send \( \text{AnsId}(\text{last}_i, \text{closed}_i, r \), \( p_j \)

Invocation \( \text{SPLITTER}(id_i) \) 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 \( \text{Check}(r) \) or \( \text{AnsId}(\ast, *, r) \) from \( p_i \) replies with a message \( (\text{AnsId}(\ast, *, *), (\text{AnsCheck}(\ast, r) \) from \( p_i \), each \( \text{wait until} \) statement (on lines 7 or 10) eventually terminates. As the number of rounds performed by \( p_i \) is bounded (by \( N = \lfloor \frac{n}{\pi - 1} \rfloor + 1 \)), it follows that \( p_i \) invocation of \( \text{SPLITTER}(id_i) \) eventually returns.

By the code, for any process \( p_i \), whenever the variables \( \text{closed} \), \( \text{last}_i \) 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 \( \text{closed} \), \( \text{last}_i \) form an increasing sequence.

Lemma 3.3. Let \( p \) be the number of processes that invoke the function \( \text{SPLITTER}() \). At most \( p - 1 \) processes return down.

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

Lemma 3.4. Let \( p \) be the number of processes that invoke the function \( \text{SPLITTER}() \). At most \( p - 1 \) processes return right.

Proof. For each process \( p_i \) whose invocation of \( \text{SPLITTER}() \) returns right, let \( r_i, 1 \leq r_i \leq N \) be the value of \( r_i \) when \( r_i \) is returned (line 8). Among the processes that return right, let \( p_m \) be a process whose associated value \( r_m \) is maximal. By the code, in the \( r_m \)-th round, \( p_m \) receives a message \( (\text{AnsCheck}, c, m, *) \) with \( c_m \geq r_m \) from some process \( p_i \), possibly \( p_m \) itself. At process \( p_i \), \( c_m \) was the value of \( \text{closed}_i \), when that message was sent. The value of \( \text{closed}_i \) is changed to \( c_m \) either when \( p_i \) performs the \( c_m \)-th round of the \( \text{for} \) loop without returning right (line 9) or when it receives a message \( (\text{AnsId}, *, *, r_i) \) from some process \( p_i \). As finitely many processes invoke the \( \text{SPLITTER}() \), a process \( p_j \) at which \( \text{closed}_i \) changed to \( c_m \) on line 9 is eventually found. This process cannot return right.

As an invocation of \( \text{SPLITTER}() \) 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 \( \text{SPLITTER}() \) and that process returns, it returns stop.

Lemma 3.6. At most \( \lfloor \frac{n}{\pi - 1} \rfloor \) processes return stop.

Proof. Let \( p_i \) be a process that terminates the \( r \)-th round (lines 8–13) without returning down or right, for some \( r \leq N \). 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 \( (\text{AnsId}, *, *, r) \) 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 \( c_i \) and \( \text{last}_i \) be the value of the variable \( \text{closed} \), and \( \text{last}_i \), respectively, at the end of round \( r \). Note that \( \text{last}_i = id_i \), since \( p_i \) does not return down (line 13).

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

Message \( m \) answers the message \( (\text{Id}, c, r, c, r) \) broadcast by \( p_i \). \( \ell_i \) is the value of \( \text{last}_i \) when this message is broadcast.

Since the successive values of \( \ell_i \) forms an increasing sequence (Observation 3.2), \( \text{last}_i = id_i \) at the beginning of the \( p_i \),
invocation of \text{splitter() (line 4)} and \text{last}_r = id_r \) at the end of round \( r \), \( \ell_r = id_r \).

When the message (Id, \( \ell_r \), s, r) from \( p_i \) is received by \( p_k \), \text{last}_k is changed to \( \ell_r \), if \( \ell_r \) is larger than the current value of \text{last}_k (line 17) and \( m \) is sent to back. The id \( \ell \) carried by \( m \) is the current value of \text{last}_k. Hence, \( \ell_r \geq \ell \), \( \ell \), (line 15).

Finally, \( id_r = \text{last}_r \geq \ell_r \) since \( \ell_r \) in \( M(\ell, r) \) and, before the end of round \( r \), \text{last}_r is changed to the largest id carried by the messages in \( M(\ell, r) \) if this id is larger than the current value of \text{last}_r. It thus follow that \( \ell_r = id_r \), Similarly, we have \( \ell_r = id_r \).

Note that \( \ell_r \) and \( \ell_r \) are the values of the local variable \text{last}_r when messages \( m \) and \( m' \) are sent, respectively. Since \( id_r > id_{r'} \), if \( m \) follows from Observation 3.2 that \( m' \) is sent before \( m \).

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

At process \( p_i \), at the end of round \( r \), for any message \((\text{AnsId}, \ast, cl, r) \in M(\ell, r) \), the value of \text{closed}_i is larger than or equal to \( c \) (line 12). It thus follows that \( c_i \geq c \geq r' \). Since \( r < N \) and \( p_i \) returns \text{stop}, \( p_i \) performs round \( r + 1 \).

As the value of \text{closed}_i is non-decreasing, \text{closed}_i \geq r + 1 in round \( r + 1 \), from which we conclude that \( p_i \) returns \text{right}; a contradiction.

\[ \square \]

For \( r, 1 \leq r < N \), let \( E_N \) denote the set of processes that terminate round \( r \) without returning either \text{down} or \text{right}.

Note that the set of processes that return \text{stop} is a subset of \( E_N \). We show that \( |E_N| \leq \lfloor \frac{n}{N} \rfloor \).

Let \( E_N = \{p_1, \ldots, p_m\} \) with \( id_1 > \ldots > id_m \), and assume for contradiction that \( m \geq \lfloor \frac{n}{N} \rfloor + 1 \). By Claim C, for all \( i, j, 1 \leq i < j \leq N \), \( \text{id}(i, j) = \emptyset \).

By definition, for all \( i, j, 1 \leq i < j \leq N \), \( Q(i, j) \) is a set of \( n - f \) process. Hence, \( \cup_{1 \leq i \leq N} Q(i, j) = N \times (n - f) > n \); a contradiction since the system consists in \( n \) processes.

\section{Long-lived splitters}

An implementation of a long-lived \( \lfloor \frac{n}{N} \rfloor \)-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 \text{right} or \text{down}, or, after returning \text{stop}, by invoking \text{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 \text{closed}_i is replaced by an array \text{Clsd}_i, as each process now closes each door on its own, adding its personal padlock. That is, \text{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 \text{splitter()}, the same door can be opened and closed by the same process, \text{Clsd}_i[id] is associated with a timestamp stored in \text{TClsd}_i[id]. When leaving the splitter or releasing it, a process \( p_i \) with identity \( id \), removes its padlocks by including its timestamp \text{TClsd}_i[id] and setting \text{Clsd}_i[id] 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 \text{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 \((\text{Clsd}_i, \text{TClsd}_i)\) (line 14).

\begin{algorithm}
\caption{Long-lived \( \lfloor \frac{n}{N} \rfloor \)-splitter (code for process \( p_i \))}
\begin{algorithmic}[1]
\STATE \textbf{1: initialization}
\STATE \( s_i \leftarrow 0 \);
\FORALL \( id \in (\text{Clsd}_i, \text{TClsd}_i, id) \leftarrow \langle 0, 0 \rangle \)
\STATE \textbf{2: function splitter(id)}
\STATE \( s_i \leftarrow s_i + 1 \)
\FORALL \( id \in (\text{Clsd}_i, \text{TClsd}_i) \leftarrow \langle r_i, \text{TClsd}_i, id \rangle + 1 \)
\STATE \textbf{3: broadcast(Check(, r_i))}
\STATE \textbf{4: wait until} \( n - f \) messages \((\text{AnsCheck(, r_i)) \}
\STATE \textbf{5: if id \in (\text{Clsd}_i, \text{TClsd}_i) \leftarrow \langle r_i, \text{TClsd}_i, id \rangle + 1 \)
\STATE \textbf{6: broadcast(Release(, C, T))}
\STATE \textbf{7: if \max(id such that C, T)} \leftarrow \langle C, T \rangle \}
\STATE \textbf{8: if id such that C, T \leftarrow \langle C, T \rangle \}
\STATE \textbf{9: release(, C, T)}
\STATE \textbf{10: return down}
\STATE \textbf{11: return stop}
\STATE \textbf{12: function release(, C, T)}
\STATE \textbf{13: while} \( m \) is received from process \( p_i \)
\STATE \textbf{14: case m = (Check(, r_s)) \textbf{do} send}
\STATE \textbf{15: \textbf{send}}
\STATE \textbf{16: function release(, C, T)}
\STATE \textbf{17: while} \( m = (\text{Check(, r_s)} \textbf{to} p_i \}
\STATE \textbf{18: \textbf{send}}
\STATE \textbf{19: function merge(S)}
\STATE \textbf{20: for all id do C, T) \leftarrow max(C, T, id) \leftarrow \langle C, T \rangle \}
\STATE \textbf{21: return(, C, T)}
\STATE \textbf{22: when a message m is received from process p_i}
\STATE \textbf{23: case m = (Check(, r_s)) \textbf{do send}}
\STATE \textbf{24: \textbf{send}}
\STATE \textbf{25: m = (Release(, C, T)) \textbf{do \text{Clsd}_i, \text{TClsd}_i = merge}}
\STATE \textbf{26: \text{Clsd}_i, \text{TClsd}_i = merge}}
\end{algorithmic}
\end{algorithm}
**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 \( \frac{1}{\alpha} \) processes.

**Lemma 4.1 (Termination).** For any correct process \( p_i \), any invocation of SPLITTER(id) 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 \( \{ \text{Clsd}[id], \text{TClsd}[id] \} \) are ordered first by the value of the timestamp \( \text{TClsd}[id] \) and then, in case of equality, by \( \text{Clsd}[id] \) (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 \( \text{Clsd}[id], \text{TClsd}[id] \) is updated, it receives a larger value. Hence,

**Observation 4.2.** For each process \( p_i \), and each identity \( id \), the successive values of \( \{ \text{Clsd}[id], \text{TClsd}[id] \} \) form an increasing sequence.

If the splitter is idle at time \( t \), then for every process and every id, \( \{ \text{Clsd}[id], \text{TClsd}[id] \} = (0, t) \) where \( t \) is the last value assigned to \( \text{TClsd}[id] \) by the process with identity \( id \). Moreover, there is no pending message. Thus, after an idle period, \( \text{Clsd}[id] = 0 \) while \( p_i \) does not change the value of \( \text{Clsd}[id] \). Hence,

**Observation 4.3.** After an idle period, \( \text{Clsd}[id] = 0 \) until \( p_i \) 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 \( P \) of all processes that invoked SPLITTER() during this period, and let us note \( p_{\text{max}} \) the process with the largest identity \( id_{\text{max}} \) in \( P \).

For a process \( p_i \) with identity \( id \), to return down, it has to verify \( \text{max}(id) \neq id \) (line 14). In other words, either \( \text{Clsd}[id] = 0 \), or \( 3id_j > id \), such that \( \text{Clsd}[id_j] > 0 \). The first case is impossible, since the \( \{ \text{Clsd}[id], \text{TClsd}[id] \} \) are in increasing order, and because the last one (overwriting others) is generated by \( p_i \) itself in line 10, with \( \text{Clsd}[id] = 0 \).

Thus, if \( p_{\text{max}} \) returns down, this means that \( 3id_j > id_{\text{max}} \) such that \( \text{Clsd}_{\text{max}}[id_j] > 0 \). Because just before the busy period, the splitter was idle, this means that, from observation 4.3, process \( p_j \) must have invoked SPLITTER() after the idle period and before the moment when \( p_{\text{max}} \) returns. Thus, process \( p_j \) has invoked SPLITTER() during this busy period, and \( id_j > id_{\text{max}} \). This contradicts the definition of \( p_{\text{max}} \).

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 \( \text{Clsd}[id] \geq r_i \) (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, \( \text{Clsd}[id] \geq r \). \( r_m \) is well defined, as for each process \( p_i \), \( r_i \leq N \).

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

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

Therefore \( inv_{\text{m}} \) 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 \( \tau_2 \) 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 \( \tau_2 \) 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 \( \tau_2 \), there is only a finite number of invocations by processes \( p_j \), \( id_j > id_m \), and thus a finite number of different \( \{ \text{Clsd}[id], \text{TClsd}[id] \} \) values.

After time \( \tau_1 \), every invocation of SPLITTER() by process \( p_m \) returns down. This can only happen if \( \text{Clsd}_{\text{m}}[id] > 0 \) 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 \( \text{Clsd}_{\text{m}}[id] = 0 \) for all \( id \). If this is not the case, right is returned. Thus, the value of \( \text{Clsd}_{\text{m}}[id] \) changes during the invocation.

We know from observation 4.2 that the values of \( \{ \text{Clsd}_{\text{m}}[id], \text{TClsd}_{\text{m}}[id] \} \) 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 \( \{ \text{Clsd}_{\text{m}}[id], \text{TClsd}_{\text{m}}[id] \} \), 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 \( \langle \text{Clsd}[id], \text{TClsd}[id] \rangle \) 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 call 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[\( id \)] = 0 for any \( id \) including \( id_i \). Thus no invocation returns right. Since \( p_i \) is the only process for which \( \text{Clsd}[id] > 0 \), \( \max(\{id : \text{Clsd}[id] > 0\}) = id_i \) is always true. Hence no invocation returns down.

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

**Proof.** Suppose for contradiction that at some time \( r, s > [n]/(n-f)\) processes have returned stop and have not yet invoked Release(). Let \( inv_1, \ldots, inv_r \) denote the last invocation of Splitter() preceding \( r \) 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 \( \text{AnsId} \) (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 \( p \in Q(r, i) \cap Q(r', i') \). \( p \) is a process that performs invocation \( inv_i \), and it follows that \( id_i \) and \( id_{i'} \) have sent these messages. A key ingredient in the proof is the following claim:

In round \( r \) of invocation \( inv_i \), \( p_i \) receives a set of \( (n-f) \) messages \( \text{AnsId} \) (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 \( p \in Q(r, i) \cap Q(r', i') \). \( p \) is a process that performs invocation \( inv_i \), and it follows that \( id_i \) and \( id_{i'} \) 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 \( p \in Q(r, i) \cap Q(r', i') \). \( p \) is a process that performs invocation \( inv_i \), and it follows that \( id_i \) and \( id_{i'} \) 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 \( p \in Q(r, i) \cap Q(r', i') \). \( p \) is a process that performs invocation \( inv_i \), and it follows that \( id_i \) and \( id_{i'} \) 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 \( p \in Q(r, i) \cap Q(r', i') \). \( p \) is a process that performs invocation \( inv_i \), and it follows that \( id_i \) and \( id_{i'} \) 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 \( id_i < id_{i'} \), \( Q(r, i) \cap Q(r', i') = \emptyset \).
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 uncontested 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 \( i \)-th k-splitter in the network is implemented by an instance \( S^i \) of some long-lived k-splitter algorithm \( S \), 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 \( i_\ell \), after accessing k-splitters \( i_1, ..., i_{\ell-1} \) 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 \( i_1, ..., i_{\ell-1} \) by \( p_i \) has returned. Hence, each k-splitters \( i_1, ..., i_{\ell-1} \) are not idle and, and hence another process \( p_j \) may obtain the same direction as \( p_i \) when accessing splitters \( i_1, ..., i_{\ell-1} \). \( p_j \) may thus invoke also SPLITTER() on the \( \ell \)-th splitter, from which it follows that another splitter \( i_{\ell+1} \) has to be accessed by \( p_j \) before it acquires a name. The argument can be repeated to extend to arbitrary lengths the sequence of splitters accessed by \( p_j \) 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 \( S \) 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 \( \sigma_i \) of \( p_i \). When such a message is received by some process \( p_j \), \( p_j \) changes its states \( \sigma_j \) 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 \( S \) consists in \( n \) local algorithms, \( S_1, ..., S_n \) one per process. As we want to implement a grid of M splitters, each process \( p_i \) executes M instances \( S^i_1, ..., S^i_n \). At each process \( p_i \), instead of executing M independent threads, one per instance, process \( p_i \) take step in each instance \( S^i_1 \) 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. \( \text{init} \)
2. \( \text{for each } \ell \in [1..n] \text{ do } \sigma_\ell[i] \leftarrow \text{initial state of } S^i_\ell; \)
3. \( \text{Ans}[1,2,..,\ell] \leftarrow [0,\ldots,0] \)
4. \( s_i \leftarrow 0; \last_send[i] \leftarrow [0,\ldots,0]; \)
5. \( \text{last_recv}[i] \leftarrow [0,\ldots,0]; \ell \leftarrow 1 \)
6. \( \text{while true do} \)
7. \( \ell \leftarrow (\ell \mod M) + 1; \text{ take a step of } S^i_\ell; \)
8. \( \text{for each } j \in [1..n] \text{ do send } (\sigma_j, \ell) \) \( \text{to } p_j \)
9. \( \text{ Ans} \leftarrow \emptyset; \text{ wait until } |\text{recv}[\ell]| \geq n-f \)
10. \( \text{when a message } m \text{ is received from process } p_j \)
11. \( \text{ case } m = (\text{Query}, (\sigma, \ell), \text{lr}, s) \text{ do} \)
12. \( \text{ if } \text{lr} < \last_send[j] \text{ then} \)
13. \( \text{ Reject, last_send}[j], s \text{ to } p_j; \)
14. \( \text{ else last_send}[j] \leftarrow \last_send[j] + 1; \text{ do } m \text{ is accepted} \)
15. \( \text{ for each } \ell \in [1..M] \text{ do update}(\sigma[i], \ell[i]); \)
16. \( \text{send } (\text{Ans}, \sigma, \last_send[i], s) \text{ to } p_j \)
17. \( \text{ case } m = (\text{Answer}, \sigma, s, \text{ls}) \text{ do} \)
18. \( \text{ last_recv}[j] \leftarrow \text{ls}; \text{Ans}[s] \leftarrow \text{Ans}[s] \cup \{m\}; \) \( \text{ for each } \ell \in [1..M] \text{ do update}(\sigma[i], \ell[i]); \)
19. \( \text{ case } m = (\text{Reject}, \text{ls}, s) \text{ do} \)
20. \( \text{ last_recv}[j] \leftarrow \text{max}(\text{ls}, \text{last_recv}[j]); \text{ if } s = s_i \text{ then send } \text{ (Query, last_qi, last_recv[j], s)} \)

When at process \( p_i \), 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 \( \text{last_send}[i] \) identifies the Answer sent by \( p_j \) to \( p_i \). At process \( p_i \), \( \text{last_recv}[j] \) 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 \( p_i \) 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 \( \sigma[i] \) of the instance \( S^i_\ell \) on behalf of which the message is sent but also
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 \( \frac{n^3}{\sqrt{D}} \)-splitters implements long-lived \( (M, \frac{n^3}{\sqrt{D}}) \)-renaming provided that the implementation of the splitter is 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 \( \text{SPLITTER}(i) \) on \( s_{i,1} \). It then traverses the grid, following the directions returned by its \( \text{SPLITTER}() \) invocation, until it gets back \( \text{STOP} \). The name acquired by \( p_i \) is the one associated with the splitter from which it gets back \( \text{STOP} \). To release the name it has previously acquired, \( p_i \) invokes \( \text{RELEASE}() \) on the corresponding splitter.

In an execution, we say that a process is in the diagonal \( d \) at time \( \tau \) if it has invoked \( \text{SPLITTER}() \) on a splitter of \( s \in d \), and that invocation has not returned by time \( \tau \). By extension, for \( d' \leq d \), a process is present in the diagonals \( [d, d'] \) at time \( \tau \) if it is in a diagonal \( d' \) at time \( \tau \), 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 \( d_2 \geq d_2 \) 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 \( \tau \), 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 \( \text{SPLITTER}() \) on splitters in diagonal \( 1, 2, \ldots, d-1 \) in that order. Let \( \tau_0 \) be the last time before \( \tau \) when a process was in diagonal \( d_1 \).

A splitter is occupied at some time \( \tau \) if an operation has been invoked on that splitter before time \( \tau \) and this operation has not returned by time \( \tau \). Let \( S \subset \cup_{d_2 \leq d} d \) denote the set of splitters that are occupied at some time between \( \tau_0 \) and \( \tau_f \). Let \( n_d \) be 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 \( \tau_0 \).

If a splitter \( s \) in \( S \) and in diagonal \( d \) \((d_1 \leq d \leq d_2)\) is idle at time \( \tau_0 \), 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 \( \text{STOP} \), or at least \( 2 \) processes access it during the period \([\tau_0, \tau_f] \). Since every process reaches at least diagonal \( d_2 \), no process gets back \( \tau \) during the interval \([\tau_0, \tau_f] \). (Otherwise, such a process returns to the initial diagonal in the interval, contradicting the definition of \( \tau_0 \)). Therefore at least \( 2 \) processes concurrently access \( s \).

Moreover, as \( s \) is idle when the first access starts, because no process may \( \text{STOP} \), 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 \( \tau_0 \), then every \( \text{SPLITTER}() \) invocation on \( s \) in the interval \([\tau_0, \tau_f] \) may return the same direction, right or down (but not \( \text{STOP} \)). Thus, at least one splitter following \( s \) in the next diagonal is in \( S \).

Then, by an induction on \( NI_d \), we prove that \( n_{d+1} - n_d \geq 1 - NI_{d+1} \). Summing these inequalities leads to \( n_{d_2} - n_{d_1} \geq d_2 - d_1 - NI_{d_1+1} \), where \( NI_{d_1+1} \) is the number of splitters in the diagonals \([d_1, d_2] \) that are non-idle at time \( \tau_0 \). As, by Lemma 6.1, \( NI_{d_1+1} \leq B \), we have \( n_{d_2} - n_{d_1} \leq d_2 - d_1 - B \leq n_{d_1} + B + B - p - B \geq p + 1 \), since at least one splitter in diagonal \( d_1 \) is occupied at \( \tau_0 \) (and thus \( n_1 \geq 1 \)).

But, during the interval \([\tau_0, \tau_f] \), 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 \( D = D_k = B + 1 + \sum_{k=1}^{D_k-1} (D_k - 1) \). If at most \( p \) processes use the grid, each process returns \( \text{STOP} \) 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 \( d \geq D_k \). By applying again Lemma 6.2 to diagonals \( d \geq D_k \), it follows that at most \( p-2 \) processes can be in the diagonals \( d \geq D_k - 1 + D_{k-1} \).

Therefore, by induction, we have that at most \( p \) processes can be in the diagonals \( d \geq D' \). At the time \( \tau \) 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 \( \tau \) becomes non-idle unless the process \( p_i \) enters it and returns \( \text{STOP} \). Thus, \( p_i \) can visit at most \( B + 1 \) different splitters before returning \( \text{STOP} \). Since there are \( B+1 \) diagonals in the diagonal \([D', D] \), then \( p_i \) necessarily returns \( \text{STOP} \).

Hence, the previous lemma implies that the grid \( G \) implement long-lived \( (M, \frac{n^3}{\sqrt{D}}) \)-renaming for \( M = O(n^4) \).

Theorem 6.4. There is a message-passing \( (M, \frac{n^3}{\sqrt{D}}) \)-renaming algorithm for \( M = O(n^4) \).
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 $\frac{n}{n-f+1}$ partitions may occur. It is thus not possible to guarantee that stop is returned to less than $\left\lfloor \frac{n}{n-f+1} \right\rfloor$ from a splitter or that new names in renaming are shared by less than $\left\lfloor \frac{n}{n-f+1} \right\rfloor$ processes. The paper has provided implementations of one-shot and long-lived $f$-tolerant implementation of $\left\lfloor \frac{n}{n-f+1} \right\rfloor$-splitters. It has also shown that, despite their weak semantic, when appropriately composed, long-lived $\left\lfloor \frac{n}{n-f+1} \right\rfloor$-splitters can be used as a basic building block to implement $(M, \left\lfloor \frac{n}{n-f+1} \right\rfloor)$-renaming, where the size of the name domain $M = O(n^k)$. Obvious direction for future research is to improve the size of the domain of new names.

8. REFERENCES


