Splitters: Objects For On-Line Partitioning*

Jaap-Henk Hoepman

Department of Computer Science, University of Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, the Netherlands
jhh@cs.kun.nl

Abstract A splitter is a concurrent asynchronous non-blocking object that can partition a collection of contending tokens into smaller groups with certain properties. Splitters are natural objects used to solve a wide range of fundamental distributed computing problems, like renaming and resource allocation. This paper proposes a general definition of splitters, develops their theory, and investigates their implementation in shared memory systems.

Keywords: splitters, shared objects, asynchronous communication, divide & conquer.

1 Introduction

Many fundamental problems in distributed computing can be solved efficiently using a divide and conquer strategy. For asynchronous systems, implementing a suitable divide and conquer strategy sometimes turns out to be a hard problem. To investigate the exact nature of this problem we are motivated to study existing concurrent objects called *splitters*, that have been used to solve this problem in the past, but never received an independent study.

A splitter is a concurrent asynchronous non-blocking object that can partition a collection of contending tokens into smaller groups with certain properties. Conceptually, a splitter has a single input over which it receives incoming tokens, and two or more outputs over which tokens leave the splitter. Each token is assigned to exactly one of the outputs, depending on the contention on the input and the distribution of tokens at the output. The specification of the splitter defines how this assignment takes place. By choosing the right specification, a splitter can be used to partition processors in roughly equal sets, or even to count the number of currently contending processors. In fact, splitters have been used for specific purposes (either implicitly or explicitly) in several distributed algorithms through the years, like mutual exclusion [Lam87], renaming [MA95, AM94, BGHM95], and resource allocation [AHS94].

Because of their occurrence in a wide variety of algorithms, and their fundamental divide-and-conquer nature, we wish to embark on a general study of these splitters in isolation, as a class of objects in their own right. Splitters are a generalisation of counting networks investigated by Aspnes *et al.* [AHS94],

* Id: splitter.tex,v 1.5 2003/11/11 10:01:33 hoepman Exp

and encompass these as well as threshold networks, balancers and smoothing networks. We refer to Sect. 3 for examples of how these (and other) objects can be defined as splitters.

In this paper we report on our initial findings of our study of splitters. Our main contribution is twofold. First of all, we have developed a general model and notation for the description of and reasoning about splitters. This model and notation is described in Sect. 2. Using some natural properties and axioms, we manage to keep this general definition surprisingly simple: almost all splitters can be described by a series of simple inequalities bounding the output contention from above. In Sect. 3 we give some examples of splitter definitions corresponding to splitter like objects used in the literature.

Some of these splitters have been implemented in the read/write memory model; others were implemented using stronger primitives like read-modify-write. In this paper we focus on the implementation of splitters in the read/write shared memory model. As our second contribution, we investigate the implementation of splitters in the read/write shared memory model, showing both impossibility results and splitter constructions. We show for instance that in a read/write memory model, splitters cannot distribute tokens over all their outputs evenly, and that no read/write implementations of non-trivial 2-output splitters exist. These results are presented in Sect. 4.

Finally, we summarise our conclusions and present topics for further research in Sect. 5.

2 Model & Notation

A splitter is a concurrent, asynchronous, non-blocking object shared among n processors[1]. Conceptually, a splitter has a single input x and one or more outputs $y_1, \ldots, y_m$. The total number of outputs is denoted by m. Processors can send one token at a time to the input of a splitter, after which the splitter will assign an output over which the token leaves the splitter (within a bounded number of steps) to join the corresponding output token set. Splitters can either be *one-shot* – in which case tokens stay in the output set forever – or be *long-lived* – in which case tokens may leave the output set travelling back through the splitter to the input.

The interface of a splitter consists of two operations: *Enter* (to enter a splitter and obtain an output) and *Release* (to return back to the input), where the *Release* operation is only defined for long-lived splitters. Tokens using[2] splitters can be in one of four states: *idle*, *entering*, *assigned* or *releasing*. Initially, tokens are *idle*. To enter a splitter S (during which it is in the *entering* state), an idle token invokes the operation $y = \mathit{Enter}(S)$, returning the selected output for this token

[1] Splitters can be realised both in the message passing as well as the shared memory model. In this paper we focus on the shared memory model.

[2] In the remainder of this paper we will assume that tokens are active objects, that maintain state and executes steps by themselves. This makes notation and discussion easier.

when it finishes. At this point a token becomes *assigned* to output y. For long-lived splitters, an assigned token may invoke the *Release*(S) operation (during which it is in the *releasing* state), after which it becomes idle again. A token is *contending* if it is not idle.

Next we define the state of a splitter.

Definition 2.1. *The state of a splitter is given by the states of all tokens that contend it, and is denote by σ. For token t we write $\sigma(t)$ for its state in σ (where $\sigma(t) = \perp$ means t is idle, $\sigma(t) = \rightarrow\!\odot$ means t is entering, $\sigma(t) = i$ means t is assigned to output y_i, and $\sigma(t) = \odot\!\rightarrow$ means t is releasing). Similarly, we write $\sigma(t) : v$ to denote the state equal to σ except that t has state v. A state is called a* steady state *if all contending tokens are assigned to an output.*

The state of a splitter can alternatively be expressed using the *contention* at the input and the outputs of the splitter. Let z denote an input or an output of a splitter S. For $z = x$ 'a token at z' means a token is contending S, while for $z = y_i$ 'a token at z' means a token assigned to output y_i. We distinguish the following four different contention measures.

point contention $\eth^t z$ of S at time t: the number of tokens at z at time t.

maximal point contention $\delta^t z$ of S at time t: the maximal number of tokens at z at any time t' within the busy prefix of S at t.

interval contention $\Delta^t z$ of S at time t: the total number of *different* tokens (i.e., not counting doubles) at z in the busy prefix of S at t (this measure is also called interval contention in [AAF+99]).

total contention $\nabla^t z$ of S at time t: the total number of tokens (counting doubles) at z in the busy prefix of S at t.

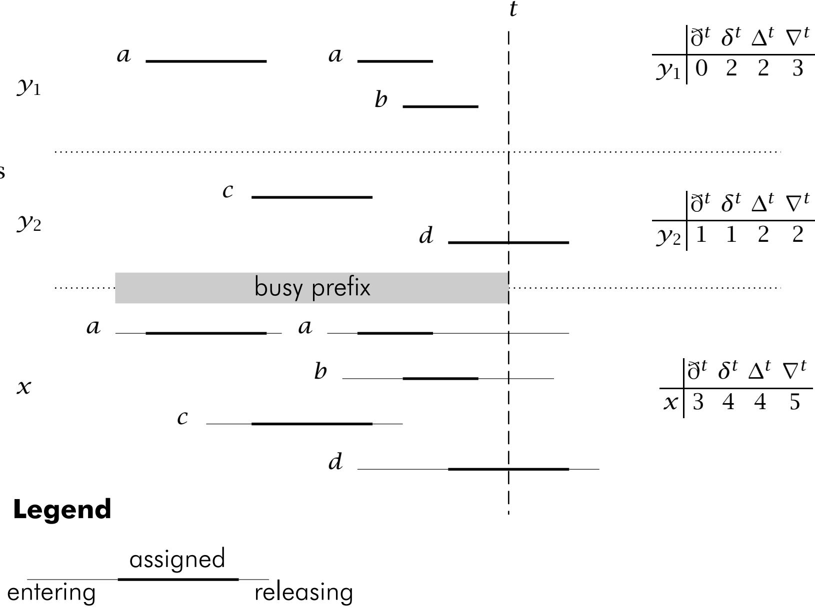
Here, the busy prefix of S at t is defined as the time interval between t and the last $t' \leq t$ where all tokens are idle on S. Note that if the same token contends more than once during a busy prefix, it contributes only once to the interval contention.

We usually omit the superscript t. Note that δy_i, Δy_i and ∇y_i are measured over the busy prefix of S, and not over the busy prefix of y_i itself. This means that if during the busy prefix of S only one token enters and leaves y_i several times (say three), then $\nabla y_i = 3$. See also Fig. 1, which shows a particular run over a 2-output splitter being accessed by 4 different tokens a, b, c and d. Note that $\eth \leq \delta \leq \Delta \leq \nabla$, and equality always holds for one-shot splitters.

The implementation of a splitter should be *adaptive*, meaning that the number steps needed to enter or release a splitter depends solely on the number of contending tokens. Note that adaptive splitters are *wait-free* by default.

The behaviour of a splitter S is defined by its invariant $\mathsf{Inv}(S)$. $\mathsf{Inv}(S)$ is a predicate over the states σ of S. We write $\sigma \vDash P$ if predicate P holds in state σ. An implementation of a splitter S must ensure that for each state σ that can occur during an execution over S, $\sigma \vDash \mathsf{Inv}(S)$ holds. Because we are interested in splitters as objects that can partition a collection of contending tokens into smaller groups, we restrict the invariant of a splitter to be a predicate over the

PSfrag replacements



	$\eth^t$	δ^t	Δ^t	∇^t
y_1	0	2	2	3

	$\eth^t$	δ^t	Δ^t	∇^t
y_2	1	1	2	2

	$\eth^t$	δ^t	Δ^t	∇^t
x	3	4	4	5

Figure 1. A run over a long-lived splitter and the resulting contentions.

input and output contentions of the splitter only. In other words, for splitter S with m outputs, $\mathsf{Inv}(S)$ is a predicate over input contention dx and output contentions dy_i, where the symbol d ranges over $\eth$, δ, Δ or ∇.

Tokens are only allowed to enter the splitter if that does not invalidate the invariant. A typical example is an invariant that restricts the maximal number of contending tokens to a constant. We only consider *smooth* long-lived splitters, where no such restriction is placed on the release operation: a token should always be able to release itself (see below).

2.1 Properties and axioms

Splitters satisfy the following properties, and are further defined by the following axioms.

Clearly, the contention is always greater or equal to 0.

Property 2.2. For any state σ and input or output z of a splitter S, we have $\eth z \geq 0$, $\delta z \geq 0$, $\Delta z \geq 0$ and $\nabla z \geq 0$.

As already stated earlier, the four contention measures form an increasing series.

Property 2.3. For any state σ and input or output z of a splitter S, we have $\eth z \leq \delta z \leq \Delta z \leq \nabla z$. Equality always holds if S is one-shot.

Proof. Follows from the fact that if a token contends at time t it also contends at the busy prefix of t, and the fact a token never contends with itself. Equality holds for one-shot splitters, because tokens never leave. □

Point contention on the outputs cannot exceed contention on the input. Similarly for total contention.

Property 2.4. For any state σ of splitter S with m outputs, $\sum_{i=1}^{m} \eth y_i \leq \eth x$ and $\sum_{i=1}^{m} \nabla y_i \leq \nabla x$, with equality holding in the steady state.

Proof. If a token is assigned to output y_i at time t it is contending at time t. For point contention we also need the fact that, no token is assigned to two outputs simultaneously. For total contention, we use the fact that all duplicates are counted. Equality in the steady state follows from the fact that no tokens are unassigned in the steady state. □

Note that this property does not hold for the interval contention (or the maximal point contention) on the outputs. Suppose for an m output splitter, the first m tokens are spread evenly over all outputs and stay there. Then the $m+1$-th token enters and leaves m times, which is assigned to a different output each time. Then during this interval $\Delta x = m + 1$ and $\Delta y_i = 2$ so $\sum_{i=1}^{m} = 2m$.

For a definition of a splitter to be meaningful, it must satisfy the following axioms. First, the initial state with no tokens contending must be a legal state.

Axiom 2.5 *Let σ be the state of splitter S with all tokens idle. Then $\sigma \models \mathsf{Inv}(S)$.*

Second, if a token has entered, it must be able to legally obtain some output.

Axiom 2.6 *For all states σ of a splitter S, if $\sigma \models \mathsf{Inv}(S)$ and for some token t we have $\sigma(t) = \ominus$, then there is an i with $1 \leq i \leq m$ such that $\sigma(t) : i \models \mathsf{Inv}(S)$.*

For long-lived splitters, similar axioms govern the behaviour of the splitter when tokens are released.

A token must always be able to release itself.

Axiom 2.7 *For all states σ of a splitter S, if $\sigma \models \mathsf{Inv}(S)$ and for some token t we have $\sigma(t) = i$ with $1 \leq i \leq m$, then $\sigma(t) : \ominus\!\!\rightarrow \models \mathsf{Inv}(S)$.*

Moreover, a releasing token must be able to leave the splitter and return to the idle state.

Axiom 2.8 *For all states σ of a splitter S, if $\sigma \models \mathsf{Inv}(S)$ and for some token t we have $\sigma(t) = \ominus\!\!\rightarrow$ then $\sigma(t) : \perp \models \mathsf{Inv}(S)$.*

We restrict our attention to *smooth* splitters that we define next.

Definition 2.9. *A splitter S with m outputs is called* smooth *if its invariant* $\mathsf{Inv}(S)$ *can be specified by a collection of $m+1$ inequalities of the form*

$$d_0 x \le f_0(\sigma)$$
$$d_i y_i \le f_i(\sigma) \text{ for all } i,\ 1 \le i \le m\ ,$$

where for each i with $0 \le i \le m$, d_i is any of the four contention measures $\eth$, δ, Δ or ∇, and each f_i is a function mapping splitter states to integers.

We note that it does not make much sense to consider non-smooth splitters, because any implementation of a splitter must be smooth anyway: if the implementation assigns any number of tokens to a particular output, then each of these tokens can be delayed indefinitely until all other tokens appear on their outputs, after which the delayed tokens are released one by one.

Note that circularity in the definition is not a problem, because the functions together specify a predicate that is either true or false in any specific state, depending on whether all inequalities are true or not in that state.

A few more observations about the form of the invariant can be made.

Because there is no difference in contention measures for one-shot splitters, we always use the total contention ∇ in the description of the invariant for one-shot splitters.

Observe that by feeding a splitter with a lot of tokens, and subsequently releasing all tokens except those at output y_i, we get $\eth y_i = \eth x$. This means we cannot even impose a condition like $\eth y_i \le \eth x - 1$. Hence, $\eth x$ is not commonly used in the definition of $\mathsf{Inv}(S)$.

Finally, note that for any constant c, if for all t we have $\eth \le c$ then also $\delta \le c$. But this only holds for constants. Clearly $\eth y_i \le \eth x$ does not imply $\delta y_i \le \eth x$ (although $\eth y_i \le \delta x$ for all t does imply $\delta y_i \le \delta x$). We conclude that also $\eth y_i$ is not commonly used in the definition of $\mathsf{Inv}(S)$, because its role can be taken by δy_i.

3 Examples

To give a feel for the class of smooth splitters, we present several splitters that have appeared in the literature (albeit under different names) using the notation we developed. We first consider one-shot splitters, and then present some long-lived splitters.

3.1 One-shot splitters

Aspnes *et al.* [AHS94] have defined several splitter-like objects in their treatment of counting networks, like a *balancer*,

$$\nabla y_1 \le \left\lceil \frac{\nabla x}{2} \right\rceil \ \wedge\ \nabla y_2 \le \left\lfloor \frac{\nabla x}{2} \right\rfloor ,$$

a *counting network*

$$\text{For all } i,\ 1 \le i \le m\colon \nabla y_i \le \left\lceil \frac{\nabla x - i + 1}{m} \right\rceil ,$$

and a *k-smoothing network*

$$\text{For all } i,\ 1 \le i \le m\colon \nabla y_i \le \min\left\{\nabla y_j \mid j \ne i\right\} + k \ .$$

Note that equality in the first two invariants above is guaranteed to hold in the steady state due to Prop. 2.4. Our definition of the k smoothing network is atypical: the original definition states that in the steady state $|\nabla y_i - \nabla y_j| \le k$ for all i, j.

Analogous to the threshold networks defined by Aspnes *et al.* [AHS94], we can define a *threshold network* for threshold w with the invariant

$$\nabla y_1 \le \left\lfloor \frac{\nabla x}{w} \right\rfloor \ \wedge \ \nabla y_2 \le \nabla x - \left\lfloor \frac{\nabla x}{w} \right\rfloor .$$

In the context of renaming, splitters have also been used extensively, like the one shot "fast-path" renaming building block [MA95, Lam87] (where y_1 corresponds to *stop*, y_2 corresponds to *right*, and y_3 corresponds to *down* as in [MA95]):

$$\nabla y_1 \le 1 \ \wedge \ \nabla y_2 \le \nabla x - 1 \ \wedge \ \nabla y_3 \le \nabla x - 1$$

3.2 Long-lived splitters

Long-lived splitters are necessary to implement long-lived renaming. Buhrman *et al.* [BGHM95] used a long lived splitter in their initial phase of a fast long-lived renaming protocol that had the following invariant

$$\text{For all } i,\ 1 \le i \le 3\colon \delta y_i \le \delta x - 1 \ .$$

Other splitters are for example the long lived "fast-path" renaming building block of Afek *et al.* [AAF+99]

$$\delta y_1 \le 1 \ \wedge \ \nabla y_2 \le \nabla x - 1 \ \wedge \ \nabla y_3 \le \nabla x - 1 \ ,$$

and the long lived "fast-path" renaming building block of Moir *et al.* [MA95]

$$\delta y_1 \le 1 \ \wedge \ \delta y_2 \le \delta x - 1 \ \wedge \ \delta y_3 \le \delta x - 1 \ .$$

Note that the Moir *et al.* splitter is more permissive than the Afek *et al.* splitter, and that in fact the implementation of the Moir *et al.* splitter given in [MA95] can reach the state $\Delta y_2 = \delta x$ in the following scenario. Token 1 enters and stops, i.e., is assigned to y_1. Then token 2 enters and goes right (i.e., is assigned to y_2), and subsequently leaves. The same happens to token 3. Then $\delta x = 2$, $\Delta y_2 = 2$ but $\delta y_2 = 1$.

We see that the difference between the Moir *et al.* splitter and the Afek *et al.* splitter (as discussed in [AAF+99]) is that the former is defined using the maximal point contention on the input, whereas the latter is defined using the total contention.

4 Constructions and impossibility results

In this section we investigate the wait-free implementation of splitters in the shared memory model. We start with a few impossibility results.

Splitters cannot distribute tokens over their outputs tightly (and evenly) if they are only implemented using read/write atomicity. This is formalised in the following theorem.

Theorem 4.1. *Let S be a splitter with $m > 1$ outputs. Suppose for some constant $c > 1$ we can select constants $c_1, \dots, c_m$ such that for all states σ of S with $dx = c$ we have*

$$f_i^S(\sigma) \le c_i$$

and

$$\sum_{i=1}^{m} c_i < c + \frac{m-1}{2} \ .$$

Then a read/write implementation of S does not exists

Proof. Consider the following construction for renaming $c > 1$ tokens, where splitter S is used in a one-shot fashion. W.l.o.g. then $dx = \nabla x$ and $dy_i = \nabla y_i$ (see Prop. 2.3).

Let the c tokens enter S. The conditions of the theorem guarantee that no more than c_i tokens leave the splitter over output y_i. Run at each output y_i a renaming algorithm (e.g., [AF00]) that renames the at most c_i incoming tokens to at most $2c_i - 1$ names. We must use distinct name-sets for each different output of the splitter. This is possible because $f_i^S(\sigma)$ is bounded a priori by c_i.

Then the total number of names assigned to the c incoming tokens is no larger than

$$\sum_{i=1}^{m} (2c_i - 1) = 2\sum_{i=1}^{m} c_i - m$$

Herlihy and Shavit [HS93] showed that wait-free renaming of c processes to less than $2c - 1$ names cannot be implemented using read/write atomicity. So this is the case if $2\sum_{i=1}^{m} c_i - m < 2c - 1$. □

Moreover, a splitter implemented using only reads and writes cannot partition a set of tokens into two non-empty sets.

Theorem 4.2. *Define $M = \{1, \dots, m\}$. Let S be a splitter with $m > 1$ outputs. Suppose there exists an index set $I \subset M$ such that for all states σ of S with $dx > 0$ we have*

$$\sum_{i \in I} f_i(\sigma) < \max(2, dx) \ \text{ and } \ \sum_{i \in M-I} f_i(\sigma) < dx \ .$$

Then a read/write implementation of S does not exist.

PSfrag replacements
x
y_1
y_2
a
b
c
d
t
ð^t δ^t Δ^t ∇^t
y_1 0 2 2 3
ð^t δ^t Δ^t ∇^t
y_2 1 1 2 2
ð^t δ^t Δ^t ∇^t
x 3 4 4 5

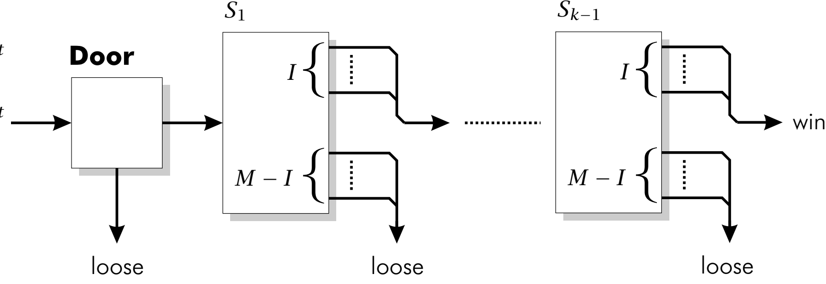


Figure 2. Construction used in the proof of Th. 4.2.

Proof. We show that if both properties hold, we can build a test-and-set object of which we know no read/write implementations exist [LAA87, Her91]. The construction is sketched in Fig. 2. Again, splitter S is used in a one-shot fashion. W.l.o.g. then $dx = \nabla x$ and $dy_i = \nabla y_i$ (see Prop. 2.3).

Let there be $k > 1$ processors for which we wish to implement a test-and-set object. The processors first have to pass through a "door" implemented using a single multi-writer shared variable DOOR, initially *open*. To pass through the door, processors execute the following protocol.

if DOOR = *closed*
then return *leave*
else DOOR ← *closed*
    return *pass*

Only processors that pass enter the following setup of splitters, each with their own token. Note that at most k tokens enter, and that at least one processor finds the door open and enters with a token.

Let some index set I satisfy the conditions of the theorem. Connect $k - 1$ copies of the splitter S as follows. Tokens leaving splitter S_i on an output y_i with $i \in I$ enter splitter S_{i+1}. Tokens leaving splitter S_i on another output (with $i \notin I$) loose the test-and-set immediately. Tokens leaving splitter S_{k-1} on output y_i with $i \in I$ win the test-and-set.

By the properties of the splitter, simple induction shows that at least one token and at most $k - i$ tokens leave splitter S_i on an output y_i with $i \in I$ (and hence enter splitter S_{i+1}). Hence at splitter S_{k-1} there is exactly one token leaving splitter S_k on output y_i with $i \in I$ and winning the test-and-set.

This only leaves us to show that any processor loosing does not strictly precede the eventual winner [Hoe99, AGTV92]. This is guaranteed by the door placed before the first splitter: it is easy to see that no processor strictly precedes (in accessing the door) any processor that passes the door. □

A splitter is trivially implemented unless $\Delta y_i < \Delta x$ for all its outputs when $\Delta x > 1$. The following corollary is a direct consequence of Th. 4.2, and completes the characterisation of the read/write implementability of splitters with 2 outputs.

Corollary 4.3. *No non-trivial splitters with* 2 *outputs have a read/write implementation.*

Next, we investigate read/write implementations of splitters with 3 outputs. Given Th. 4.2, the best we can hope for is a splitter like

$$\begin{aligned} & \nabla y_1 \le f_1(\sigma) \\ \wedge \quad & \nabla y_2 \le \nabla x - f_1(\sigma) \\ \wedge \quad & \nabla y_3 \le \nabla x - f_1(\sigma) \end{aligned}$$

where $f_1(\sigma) \le \frac{1}{2}\nabla x$.

Theorem 4.4. *Splitter S defined by*

$$\delta y_i \le \frac{2}{3}\delta x, \quad \textit{for}\ 1 \le i \le 3.$$

has a read/write implementation.

Proof. Use any optimal long-lived renaming algorithm (like [AF00]) to rename the δx incoming tokens to $2\delta x - 1$ names. Map a token with name i to output $y_{(i \bmod 3)+1}$. Then $\delta y_i \le \frac{2}{3}\delta x$.

In a read-modify-write setting, any reasonable splitter can be implemented.

Theorem 4.5. *Let S be a splitter satisfying the axioms in Sect. 2.1 shared with n processors. This splitter can be implemented using a single n processor read-modify-write register.*

Proof. Observe that the state of a splitter as given in Def. 2.1, is a set of token states. We let the read-modify-write register store this set of states (in the most general case the size of the register must be unbounded).

Initially, we let the register denote the empty set. Observe, that by Ax. 2.5, this initialisation of the register is proper.

An entering token reads the register and selects any output y_i of his choice that satisfies $\mathsf{Inv}(S)$ (such an output always exist by the Ax. 2.6). A leaving token reads the register and changes its own state to idle and writes it back. This is also a correct action, due to Ax. 2.7 and Ax. 2.8. □

5 Conclusions and further research

We have presented a general definition of splitters as a concurrent asynchronous non-blocking object that can partition a collection of contending tokens into smaller groups. This general definition turns out to be surprisingly simple: almost all splitters can be described by a series of simple inequalities bounding the output contention from above.

Any splitter can be constructed using read-modify-write registers. For the case where only read/write registers can be used, we have presented some constructions and some impossibility results. These results can be extended. In particular, there are still gaps between our upper and lower bounds. Moreover, it is an interesting question to give characterisations of splitters in terms of their place in Herlihy's hierarchy [Her91].

Also, we would like to investigate the construction of larger splitters from smaller ones, similar to the way balancers are used to construct counting networks [AHS94].

References

[AAF+99] Afek, Y., Attiya, H., Fouren, A., Stupp, G., and Touitou, D. Long-lived renaming made adaptive. In *18th PODC* (Atlanta, GA, USA, 1999), ACM Press, pp. 91–103.

[AGTV92] Afek, Y., Gafni, E., Tromp, J., and Vitányi, P. M. B. Wait-free test-and-set. In *6th WDAG* (Haifa, Israel, 1992), A. Segall and S. Zaks (Eds.), LNCS 647, Springer-Verlag, pp. 85–94.

[AM94] Anderson, J. H., and Moir, M. Using k-exclusion to implement resilient, scalable shared objects. In *13th PODC* (Los Angeles, CA, USA, 1994), ACM Press, pp. 141–150.

[AHS94] Aspnes, J., Herlihy, M., and Shavit, N. Counting networks. *J. ACM* **41**, 5 (1994), 1020–1048.

[AF00] Attiya, H., and Fouren, A. Polynomial and adaptive long-lived $(2k-1)$-renaming. In *14th DISC* (Toledo, Spain, 2000), M. Herlihy (Ed.), LNCS 1914, Springer, pp. 149–163.

[BGHM95] Buhrman, H., Garay, J. A., Hoepman, J.-H., and Moir, M. Long-lived renaming made fast. In *14th PODC* (Ottawa, Ont., Canada, 1995), ACM Press, pp. 194–203.

[Her91] Herlihy, M. P. Wait-free synchronization. *ACM Trans. Prog. Lang. & Syst.* **13**, 1 (1991), 124–149.

[HS93] Herlihy, M. P., and Shavit, N. The asynchronous computability theorem for t-resilient tasks. In *25th STOC* (San Diego, CA, USA, 1993), ACM Press, pp. 111–120.

[Hoe99] Hoepman, J.-H. Long-lived test-and-set using bounded space. Tech. rep., University of Twente, 1999. www.cs.kun.nl/˜jhh/publications/test-and-set.ps.

[Lam87] Lamport, L. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.* **5**, 1 (1987), 1–11.

[LAA87] Loui, M. C., and Abu-Amara, H. H. Memory requirements for agreement among unreliable asynchronous processes. In *Advances in Computing Research*, F. P. Preparata (Ed.), vol. 4. JAI Press, Greenwich, CT, 1987, pp. 163–183.

[MA95] Moir, M., and Anderson, J. H. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming* **25**, 1 (1995), 1–39.