Available online at www.sciencedirect.com

SCIENCE DIRECT®

Theoretical Computer Science 333 (2005) 373–400

Theoretical Computer Science

www.elsevier.com/locate/tcs

The cost of concurrent, low-contention Read&Modify&Write ☆

Costas Busch[a,1], Marios Mavronicolas[b,*,2], Paul Spirakis[c,d]

[a] Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY 12180, USA
[b] Department of Computer Science, University of Cyprus, Nicosia CY-1678, Cyprus
[c] Department of Computer Engineering and Informatics, University of Patras, Rion, 265 00 Patras, Greece
[d] Research and Academic Computer Technology Institute, P.O. Box 1122, 261 10 Patras, Greece

Received 8 October 2003; received in revised form 25 March 2004; accepted 15 April 2004

Abstract

This work addresses the possibility or impossibility, and the corresponding costs, of devising *concurrent, low-contention* implementations of atomic Read&Modify&Write (or RMW) *operations* in a distributed system. A natural class of *monotone* RMW operations associated with *monotone groups*, a certain class of algebraic groups introduced here, is considered. The popular Fetch&Add and Fetch&Multiply operations are examples from the class.

A *Monotone Linearizability Lemma* is proved and employed as a chief combinatorial instrument in this work; it establishes inherent ordering constraints of *linearizability* for a certain class of executions of *any* distributed system implementing a monotone RMW operation.

☆A preliminary version of this work appears in the *Proceedings of the 10th International Colloquium on Structural Information and Communication Complexity* (Umeå, Sweden, June 2003), J.F. Sibeyn ed., pp. 57–72, Proceedings in Informatics 17, Carleton Scientific, 2003. This work has been partially supported by the IST Program of the European Union under contract numbers IST-1999-14186 (ALCOM-FT) and IST-2001-33116 (FLAGS), by funds from the Joint Program of Scientific and Technological Collaboration between Greece and Cyprus, by the Greek General Secretariat for Research and Technology, and by research funds from Rensselaer Polytechnic Institute and University of Cyprus.

*Corresponding author.

E-mail addresses: buschc@cs.rpi.edu (C. Busch), mavronic@ucy.ac.cy (M. Mavronicolas), spirakis@cti.gr (P. Spirakis).

[1] Part of the work of this author was performed while visiting Department of Computer Science, University of Cyprus.

[2] Part of the work of this author was performed while visiting Faculty of Computer Science, Electrical Engineering and Mathematics, University of Paderborn.

doi:10.1016/j.tcs.2004.04.018

The end results of this work specifically apply to implementations of (monotone) RMW operations that are based on *switching networks*, a recent class of concurrent, low-contention data structures that generalize *counting networks* (J. ACM 41(5) (1994) 1020–1048) (which implemented the traditional Fetch&Increment operation). These results are negative; they are shown through the *Monotone Linearizability Lemma*. In particular, the *first* lower bounds on *size* (the number of *switches* in the network) for any (non-trivial) switching network implementing a monotone RMW operation are derived. It is proven that if the network incurs low contention, then its size must be infinite, no matter whether the number of states of each switch is finite or infinite. Since Fetch&Increment is implementable with counting networks of *finite*-size (J. ACM 41(5) (1994) 1020–1048), these lower bounds imply a space complexity separation between Fetch&Increment and any monotone RMW operation in the model of switching networks.

The presented lower bounds provide a mathematical explanation for the observed inability of researchers over the last thirteen years to extend counting networks, while keeping their *finite-size*, high-concurrency and low-contention, in order to perform tasks more complex than Fetch&Increment but yet as simple as Fetch&Add.

Keywords: Distributed computing; Synchronization; Linearizability; Monotone Linearizability Lemma; Switching networks; Lower bounds

1. Introduction

1.1. Background, motivation and framework

A Read&Modify&Write *shared variable* or *register* [8,11], henceforth abbreviated as RMW, is an abstract variable type that allows reading its old value, computing via some specific *operator* a new value as a function of the old one, and writing the new value back, all in a single, *atomic* (indivisible) RMW operation. For example, a Fetch&Increment register provides an operation that atomically adds one to its value and returns its prior value; a Fetch&Add register provides an operation that adds any arbitrary integer to its value and returns its prior value, while a Fetch&Multiply register does a corresponding thing for multiplication.

Most RMW operations provide strong synchronization primitives that allow for the design of efficient and transparent algorithms in the asynchronous shared memory model of distributed computation. So, it is desirable to devise suitable *distributed data structures* for the construction of highly concurrent, low-contention implementations of RMW registers. Intuitively, the *contention* of an implementation measures the extent to which concurrent *processes* access the same memory location simultaneously; it has been argued that contention is a critical factor for the overall efficiency of (asynchronous) shared memory algorithms (see, e.g., [4] and references therein). A *counting network* [2] is a particular class of *finite-size* distributed data structures used to construct high-concurrency and low-contention implementations of RMW registers that simultaneously support the Fetch&Increment and Fetch&Decrement operations [1].

The fundamental question that has motivated this work is the possibility or impossibility, and the corresponding incurred costs, of devising distributed data structures to construct highly concurrent, low-contention implementations of *general* RMW registers. In particular, is there, and at what costs, a generalization of counting networks to implement the *general* RMW operation while still retaining the nice properties of finite-size and low-contention?

We focus on a specific class of RMW operations whose operators correspond to a certain class of algebraic groups introduced and studied here, which we call *monotone groups*. A monotone group has a *total order* and a *monotone subdomain*; the latter enjoys a significant monotonicity property, which we call *Monotonicity under Composition*: applying the operator on an element from the monotone subdomain results to another element in the monotone subdomain that strictly dominates the first with respect to the total order. For example, the Fetch&Add operation clearly falls into the context of monotone groups; so also does the Fetch&Multiply operation, and so on. A *monotone* RMW operation is one that is associated with a monotone group.

We consider switching networks [6,7], a class of distributed data structures that may be used for concurrent, low-contention implementations of RMW registers; these are natural generalizations of *counting networks* [2]. Roughly speaking, a *switching network* is a directed, acyclic graph made up of nodes called *switches* and *output registers*, and edges called *wires*. A process issuing a RMW operation shepherds a *token* through the network; the token traverses a path of switches till it is eventually returned a value upon exiting the network. The *size* of a switching network is the total number of switches in it; its *concurrency* is the maximum number of concurrent *processes* that may simultaneously shepherd a token through the network.

In order to model the low-contention property for switching networks, we introduce *register bottleneck* and *switch bottleneck*; roughly speaking, both measure the *minimum* number of network elements (either output registers or switches) that are accessed by processes in any infinite execution. Intuitively, if this number is small, some element will become a *bottleneck* in some infinite execution, and the network incurs high contention; hence, a switching network is *low-contention* if register bottleneck and switch bottleneck are sufficiently large.

1.2. Contribution and significance

Our chief combinatorial instrument is a *Monotone Linearizability Lemma* (Proposition 5.1). This establishes inherent ordering constraints of *linearizability* [10] for a certain class of executions of *any* distributed system that implements a monotone RMW operation. Recall that an execution is *linearizable* [10] if the values returned to operations respect their real-time ordering.

The end results of our study are *negative*; they are shown through a modular use of the *Monotone Linearizability Lemma*. These results are the *first* lower bounds on size for any highly concurrent, low-contention switching network that implements a monotone RMW operation. For any such switching network (other than the trivial single-switch one), we prove:

- If each switch has a *finite* number of states, then the network must contain an *infinite* number of switches, even if concurrency is restricted to remain *bounded* (Theorem 6.1).

- If each switch has an *infinite* number of states, then the network must still contain an *infinite* number of switches if concurrency is now allowed to grow unbounded (Theorem 6.2).

Our impossibility results settle to the negative the general question about the possibility of devising distibuted, low-contention data structures of *finite*-size, as suitable extensions to counting networks, to support synchronization operations other than Fetch&Increment (originally supported by counting networks). This question was already stated in the seminal work of Aspnes et al. [2] that introduced counting networks; however, it has remained tantalizingly open, and progress on it has been so far limited to discovering that counting networks themselves can also support Fetch&Decrement (simultaneously with Fetch&Increment) [1]. Our results imply a space complexity separation between Fetch&Increment and any monotone RMW operation in the model of switching networks.

In summary, our lower bounds imply that we *cannot* conveniently generalize counting networks, while still retaining their finite-size, high-concurrency and low-contention, in order to perform tasks more complex than just incrementing a counter by one but yet as simple as adding an arbitrary value to a counter. Thus, our lower bounds provide a mathematical explanation for the observed inability of researchers in the last thirteen years or so (since the original conference publication of counting networks [2] in *STOC 1991*) to achieve such generalizations.

Finally, we remark that linearizability has so far been studied as a *required* property for a distributed system that best guarantees acceptable concurrent behavior. To the best of our knowledge, our work is the *first* to provide, through the *Monotone Linearizability Lemma*, an (non-trivial) instance of a distributed system where linearizability is an *inherent* property.

1.3. Related work and comparison

A particular switching network, called *Read–Modify–Write network*, is given in [7, Section 4] that implements any general class of commutative functions; Fetch&Add and Fetch&Multiply are two particular examples of such classes. This Read–Modify–Write network contains an *infinite* number of switches, and it has the same topology as a corresponding linearizable counting network presented in [9]. The *latency* (maximum number of switches traversed by a token) of this network is shown to be $O(n)$ [7, Theorem 4.14], while a corresponding lower bound of $\Omega(n)$ is also shown in [7, Theorem 3.2] for any general class of functions with certain functional properties; this family encompasses both Fetch&Add and Fetch&Multiply as special cases. In contrast, we deal, in this work, exclusively with the *size* of switching networks.

A counting network is *linearizable* [9] if the values returned to tokens respect their real-time orderings. Herlihy et al. [9, Theorem 5.1] show that any non-trivial (non-blocking) *linearizable counting network* must have infinite size. The structure of the proofs of our impossibility results is inspired by that of the proof of [9, Theorem 5.1]. The requirement that *all* executions be linearizable allows that proof to pick any arbitrary execution of choice and force it to violate linearizability. Since a switching network for a monotone RMW operation need not guarantee linearizability in all executions. The role of the *Monotone Linearizability Lemma* is to contribute executions that are *necessarily* linearizable.

1.4. Road map

Section 2 introduces monotone groups. Definitions for the model of a distributed system appear in Section 3. Section 4 provides a framework for switching networks. The *Monotone Linearizability Lemma* is the subject of Section 5. Lower bounds on the size of switching networks implementing monotone groups are shown in Section 6. We conclude in Section 7.

2. Monotone groups

In this section, we introduce and study monotone groups. We assume familiarity of the reader with the very basic concepts from Group Theory, such as a *group* $\langle \mathbb{\Gamma}, \oplus \rangle$ and an *Abelian group*. Denote e the *identity element* of the group $\langle \mathbb{\Gamma}, \oplus \rangle$. An elementary property of groups will be used in some of our later proofs is the *Cancellation Law*. It states that for any group $\langle \mathbb{\Gamma}, \oplus \rangle$, for any triple of elements $a, b, c \in \mathbb{\Gamma}$, $a \oplus b = a \oplus c$ (resp., $b \oplus a = c \oplus a$) implies $b = c$.

Throughout this section (and in the rest of the paper), denote Z, $\mathbb{N}$ and Q the sets of integers, natural numbers (including zero), and rational numbers (excluding zero), respectively. We will use $+$ and $\cdot$ to denote the common (binary) operators of addition and multiplication, respectively, on these sets. Denote $\leqslant$ the *less-than-or-equal* relation (total order) on these sets.

Some composite operators are introduced in Section 2.1. Section 2.2 provides the basic definitions for monotone groups. Section 2.3 treats n-wise independence.

2.1. Composite operators

We define two composite operators by applying the operator $\oplus$ a number of times. For any integer k, define the unary operator $\bigoplus_k : \mathbb{\Gamma} \to \mathbb{\Gamma}$ as follows:

$$\bigoplus_k a = \begin{cases} \underbrace{a \oplus a \oplus \cdots \oplus a}_{k \text{ times}} & \text{if } k > 0, \\ e & \text{if } k = 0, \\ \underbrace{a^{-1} \oplus a^{-1} \oplus \cdots \oplus a^{-1}}_{-k \text{ times}} & \text{if } k < 0. \end{cases}$$

Call $\bigoplus_k$ the *power operator*. It follows that for any element $a \in \mathbb{\Gamma}$ and integer k, $\bigoplus_k a = \bigoplus_{-k} a^{-1}$. We continue to state two elementary properties of the power operator that will be used later; their proofs are omitted as straightforward.

Property 2.1 (*Superposition of powers*). *For any Abelian group $\langle \mathbb{\Gamma}, \oplus \rangle$, fix any element $a \in \mathbb{\Gamma}$. Then, for any sequence of integers $k_1, k_2, \ldots, k_n$,*

$$\left(\bigoplus_{k_1} a\right) \oplus \left(\bigoplus_{k_2} a\right) \oplus \cdots \oplus \left(\bigoplus_{k_n} a\right) = \bigoplus_{\sum_{i=1}^{n} k_i} a.$$

Property 2.2 (*Composition of powers*). *For any group $\langle \mathbb{\Gamma}, \oplus \rangle$, fix any element $a \in \mathbb{\Gamma}$. Then, for any integer k and natural number l, $\bigoplus_k \left(\bigoplus_l a\right) = \bigoplus_{k \cdot l} a$.*

For any integer n, the operator $\biguplus_n : \mathbb{\Gamma}^n \to \mathbb{\Gamma}$ is n-ary.

- For $n = 0$, it assumes the constant value $\biguplus_0 = e$.
- For $n = 1$, $\biguplus_1 \{a\} = a$ for all elements $a \in \mathbb{\Gamma}$. For $n = -1$, $\biguplus_{-1} \{a\} = a^{-1}$.
- For $|n| \geqslant 2$. $\biguplus_n$ takes as input an ordered multiset of elements $\{a_1, a_2, \ldots, a_{|n|}\} \in \mathbb{\Gamma}$, and it yields the result
$$\biguplus_n \{a_1, a_2, \ldots, a_n\} = \begin{cases} a_1 \oplus a_2 \oplus \cdots \oplus a_{|n|} & \text{if } n \geqslant 2, \\ a_1^{-1} \oplus a_2^{-1} \oplus \cdots \oplus a_{|n|}^{-1} & \text{if } n \leqslant -2 \end{cases}$$
denoted also as $\biguplus_{i=1}^{n} a_i$. Note that, by associativity, the result of applying the operator is well defined.

Call $\bigoplus$ the *summation operator*. Our definitions for the power and summation operators immediately imply that for any element $a \in \mathbb{\Gamma}$ and for any integer $n \neq 0$,

$$\bigoplus_n a = \begin{cases} \biguplus_n \left\{ \underbrace{a, a, \ldots, a}_{n \text{ times}} \right\} & \text{if } n > 0, \\ \biguplus_n \left\{ \underbrace{a^{-1}, a^{-1}, \ldots, a^{-1}}_{-n \text{ times}} \right\} & \text{if } n < 0. \end{cases}$$

So, roughly speaking, the power operator is some special case of the summation operator where all inputs are identical. The result $\biguplus_n \{a_1, a_2, \ldots, a_n\}$ of the summation operator will sometimes be called a *composite expression*.

2.2. Monotone groups

Assume now that the set $\mathbb{\Gamma}$ is totally ordered; thus, a *total order* $\preccurlyeq$ is defined on $\mathbb{\Gamma}$. For any pair of elements $a, b \in \mathbb{\Gamma}$, write $a \prec b$ (and, equivalently, $b \succ a$) if $a \preccurlyeq b$ and $a \neq b$.

A *monotone subdomain* of $\mathbb{\Gamma}$ is a subset $\mathbb{M} \subseteq \mathbb{\Gamma}$ that satisfies the following three properties:

1. *Closure:* For any two elements $a, b \in \mathbb{M}$, $a \oplus b \in \mathbb{M}$.
2. *Identity Lower Bound:* For any element $a \in \mathbb{M}$, $e \prec a$.
3. *Monotonicity under Composition:* For any pair of elements $a, b \in \mathbb{M}$, both $a \prec a \oplus b$ and $b \prec a \oplus b$.

Notice that the *Identity Lower Bound* property implies that $e \notin \mathbb{M}$, so that $\mathbb{M} \subset \mathbb{\Gamma}$. Notice also that the *Monotonicity under Composition* property implies that $\mathbb{M}$ is necessarily infinite. A *monotone group* is a quadruple $\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \preccurlyeq \rangle$, where $\langle \mathbb{\Gamma}, \oplus \rangle$ is an Abelian group, $\preccurlyeq$ is a total order on $\mathbb{\Gamma}$, and $\mathbb{M}$ is a monotone subdomain of $\mathbb{\Gamma}$.

We encourage the reader to verify that both quadruples $\langle \mathrm{Z}, \mathbb{N} \setminus \{0\}, +, \leqslant \rangle$ (called *Integers with Addition*) and $\langle \mathrm{Q}, \mathbb{N} \setminus \{0, 1\}, \cdot, \leqslant \rangle$ (called *Rationals with Multiplication*) are monotone groups. They are associated with the monotone Fetch&Add and Fetch&Multiply operations, respectively. There follows an elementary, non-idempotency property of monotone groups that will be used in some of our later proofs.

Property 2.3 (*No idempotent power*). *For any arbitrary monotone group $\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \preccurlyeq\rangle$, fix any element $a \in \mathbb{M}$. Then, for any integer k, $\bigoplus_k a = e$ implies $k = 0$.*

The proof of Property 2.3 is straightforward; it is left as an exercise for the reader. We only remark that Property 2.3 does *not* necessarily hold for a *general* group; so, it is no coincidence that its proof relies on using the *Monotonicity under Composition* property that holds specifically for monotone groups.

2.3. *n*-Wise independence

Fix any integer $n \geqslant 2$. Consider any n distinct elements $a_1, a_2, \ldots, a_n \in \mathbb{\Gamma}$ with $a_1, a_2, \ldots, a_n \neq e$. Say that $a_1, a_2, \ldots, a_n$ are *n-wise independent over* $\langle \mathbb{\Gamma}, \oplus\rangle$ if for any sequence of n integers $k_1, k_2, \ldots, k_n$, where $-1 \leqslant k_i \leqslant 2$ for $1 \leqslant i \leqslant n$, that are *not all simultaneously* zero, $\biguplus_{i=1}^{n} \bigoplus_{k_i} a_i \neq e$. Say that the monotone group $\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \preccurlyeq\rangle$ is *n-wise independent* if there are n distinct elements $a_1, a_2, \ldots, a_n \in \mathbb{M}$, with $a_1, a_2, \ldots, a_n \neq e$, that are n-wise independent over $\langle \mathbb{\Gamma}, \oplus\rangle$.

From the definition of n-wise independence, n integers $a_1, a_2, \ldots, a_n \in \mathrm{Z}$, where $n \geqslant 2$, are n-wise independent over $\langle \mathrm{Z}, +\rangle$ if for any sequence of n integers $k_1, k_2, \ldots, k_n \in \{-1, 0, 1, 2\}$, which are not all simultaneously zero, $\sum_{i=1}^{n} k_i \cdot a_i \neq 0$. We prove.

Lemma 2.4. *For any integer $n \geqslant 2$, the monotone group $\langle \mathrm{Z}, \mathbb{N} \setminus \{0\}, +, \leqslant\rangle$ is n-wise independent.*

Proof. Fix any integer $\ell \geqslant 0$. Consider the n natural numbers $2^{\ell}, 2^{\ell+2}, \ldots, 2^{\ell+2(n-1)} \in \mathbb{N} \setminus \{0\}$, which are powers of two; we will prove that these n natural numbers are n-wise independent over $\langle \mathrm{Z}, +\rangle$. The proof is by induction on n.

For the basis case where $n = 2$, consider the natural numbers 2^{ℓ} and $2^{\ell+2}$. Fix any pair of integers $k_1, k_2 \in \{-1, 0, 1, 2\}$ that are not both simultaneously zero. Clearly, $k_1 2^{\ell} + k_2 2^{\ell+2} = 2^{\ell}(k_1 + 4k_2)$, which can be zero only if $k_1 = k_2 = 0$. So, the natural numbers $2^{\ell}, 2^{\ell+2} \in \mathbb{N} \setminus \{0\}$ are 2-wise independent over $\langle \mathrm{Z}, +\rangle$. Hence, the monotone group $\langle \mathrm{Z}, \mathbb{N} \setminus \{0\}, +, \leqslant\rangle$ is 2-wise independent. This completes the proof of the basis case.

Assume inductively that the $n - 1$ natural numbers $2^{\ell}, 2^{\ell+2}, \ldots, 2^{\ell+2((n-1)-1)} = 2^{\ell+2(n-2)} \in \mathbb{N} \setminus \{0\}$ are $(n - 1)$-wise independent over $\langle \mathrm{Z}, +\rangle$.

For the induction step, we will show that the n natural numbers $2^{\ell}, 2^{\ell+2}, \ldots, 2^{\ell+2(n-1)}$ are n-wise independent in $\langle \mathrm{Z}, +\rangle$. Assume, by way of contradiction, that they are not. Thus, there exist n integers $k_1, k_2, \ldots, k_n \in \{-1, 0, 1, 2\}$ which are not all simultaneously zero, such that $\sum_{i=1}^{n} k_i 2^{\ell+2(i-1)} = 0$. We proceed by case analysis on the value of $k_n \in \{-1, 0, 1, 2\}$.

- Assume first that $k_n = -1$. Then, $\sum_{i=1}^{n-1} k_i 2^{\ell+2(i-1)} - 2^{\ell+2(n-1)} = 0$, or $\sum_{i=1}^{n-1} k_i 2^{\ell+2(i-1)} = 2^{\ell+2(n-1)}$, or $\sum_{i=1}^{n-1} k_i 2^{2(i-1)} = 2^{2(n-1)}$. However, since $k_i \leqslant 2$ for all indices i, $1 \leqslant i \leqslant n-1$,

$$\begin{aligned}\sum_{i=1}^{n-1} k_i 2^{2(i-1)} &\leqslant 2 \sum_{i=1}^{n-1} 2^{2(i-1)} \\ &< 2 \cdot \sum_{i=0}^{2n-4} 2^{i}\end{aligned}$$

$$= 2\left(2^{2n-3} - 1\right) < 2^{2n-2} = 2^{2(n-1)},$$

a contradiction.

- Assume now that $k_n = 0$. Then, $\sum_{i=1}^{n-1} k_i 2^{\ell+2(i-1)} = 0$. Since the integers $k_1, k_2, \ldots, k_n$ are not all simultaneously zero while $k_n = 0$, it follows that the integers $k_1, k_2, \ldots, k_{n-1}$ are not all simultaneously zero. This implies that the $n-1$ natural numbers $2^{\ell}, 2^{\ell+2}, \ldots, 2^{\ell+2(n-2)}$ are $(n-1)$-wise independent over $\langle \mathrm{Z}, + \rangle$, which contradicts the induction hypothesis.
- Assume finally that $k_n \in \{1, 2\}$. Then, $\sum_{i=1}^{n-1} k_i 2^{\ell+2(i-1)} + k_n \cdot 2^{\ell+2(n-1)} = 0$, or, equivalently, $-\sum_{i=1}^{n-1} k_i 2^{\ell+2(i-1)} = k_n \cdot 2^{\ell+2(n-1)}$, or $-\sum_{i=1}^{n-1} k_i 2^{2(i-1)} = k_n \cdot 2^{2(n-1)}$. However, since $k_i \geqslant -1$ for all indices i, $1 \leqslant i \leqslant n-1$,

$$\begin{aligned} -\sum_{i=1}^{n-1} k_i 2^{2(i-1)} &\leqslant \sum_{i=1}^{n-1} 2^{2(i-1)} \\ &< \sum_{i=0}^{2n-4} 2^{i} \\ &= 2^{2n-3} - 1 < 2^{2n-2} = 2^{2(n-1)} \leqslant k_n \cdot 2^{2(n-1)}, \end{aligned}$$

a contradiction.

Since we obtained a contradiction in all possible cases, the proof is now complete. □

We finally prove that *every* monotone group is n-wise independent.

Lemma 2.5 (*Every monotone group is n-wise independent*). *For any integer $n \geqslant 2$, the monotone group $\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \preccurlyeq \rangle$ is n-wise independent.*

Proof. Since the monotone group $\langle \mathrm{Z}, \mathbb{N} \setminus \{0\}, +, \leqslant \rangle$ is n-wise independent (Lemma 2.4), there exist n distinct natural numbers $l_1, l_2, \ldots, l_n \in \mathbb{N} \setminus \{0\}$ that are n-wise independent over $\langle \mathrm{Z}, + \rangle$. Fix any element $a \in \mathbb{M}$ and consider the n elements $\bigoplus_{l_1} a, \bigoplus_{l_2} a, \ldots, \bigoplus_{l_n} a$ of $\mathbb{M}$. Clearly, by the *Monotonicity under Composition* property of the monotone group $\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \preccurlyeq \rangle$, these n elements are distinct. We will prove that they are also n-wise independent over $\langle \mathbb{\Gamma}, \oplus \rangle$.

Assume, by way of contradiction, that the elements $\bigoplus_{l_1} a, \bigoplus_{l_2} a, \ldots, \bigoplus_{l_n} a$ are *not* n-wise independent over $\langle \mathbb{\Gamma}, \oplus \rangle$. Thus, there exist n integers $k_1, k_2, \ldots, k_n \in \{-1, 0, 1, 2\}$, which are not all simultaneously zero, such that $\biguplus_{i=1}^{n} \left(\bigoplus_{k_i} \left(\bigoplus_{l_i} a \right) \right) = e$. By Property 2.2, it follows that $\biguplus_{i=1}^{n} \left(\bigoplus_{k_i \cdot l_i} a \right) = e$ which, by the definition of the summation operator, may be written as $\left(\bigoplus_{k_1 \cdot l_1} a \right) \oplus \left(\bigoplus_{k_2 \cdot l_2} a \right) \oplus \cdots \oplus \left(\bigoplus_{k_n \cdot l_n} \right) = e$. By Property 2.1, it follows that $\bigoplus_{\sum_{i=1}^{n} k_i \cdot l_i} a = e$. Property 2.3, now implies that $\sum_{i=1}^{n} k_i \cdot l_i = 0$. Since the integers k_i, $1 \leqslant i \leqslant n$, are from the set $\{-1, 0, 1, 2\}$, and they are not all simultaneously zero, this implies that the n natural numbers $l_1, l_2, \ldots, l_n$ are *not* n-wise independent over $\langle \mathrm{Z}, + \rangle$. A contradiction. □

We remark that the proof of Lemma 2.5 employs the n-wise independence of the monotone group $\langle \mathrm{Z}, \mathbb{N} \setminus \{0, \}, +, \leqslant \rangle$ (which was established in Lemma 2.4) in order to conclude the

n-wise independence of the arbitrary monotone group $\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \leqslant \rangle$. So, this proof by reduction indicates some kind of completeness of this group for the class of all monotone groups.

3. System model

Section 3.1 provides basic definitions for a distributed system that implements a monotone group. Definitions related to linearizability are given in Section 3.2.

3.1. Distributed systems implementing monotone groups

Our model of a distributed system is patterned after the one in [10, Section 2]; however, that one is adjusted in order to incorporate the issue of implementing a monotone group $\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \preccurlyeq \rangle$.

We consider a distributed system **P** consisting of a collection of sequential threads of control, called *processes*. Processes are sequential, and each process applies a sequence of operations to a distributed data structure, called the *object*, alternately issuing an invocation and then receiving the associated response. Each *invocation* at process p_i has the form $\texttt{Invoke}_i(a)$ for some value $a \in \mathbb{M}$; each *response* at process p_i has the form $\texttt{Response}_i(b)$ for some value $b \in \mathbb{M} \cup \{e\}$.

Formally, an *execution* of system **P** is a (possibly infinite) sequence α of *invocation* and *response* events. We assume that for each invocation at process p_i in execution α, there is a later response in α that matches it and no invocation at p_i that precedes the matching response in α. Prefixes and suffixes of an execution are defined in the natural way. Say that an execution γ *extends* a prefix β of execution α if β is a prefix of γ as well.

An *operation* at process p_i in execution α is a matching pair $op_i = [\texttt{Invoke}_i(a), \texttt{Response}_i(b)]$ of an invocation and response at p_i; we will sometimes say that op_i is of *type* a. For such an operation, we will write $a = \mathsf{In}(op_i)$ and $b = \mathsf{Out}(op_i)$; thus, op_i has *input* and *output* a and b, respectively. We will sometimes write $\mathsf{In}_\alpha(op_i)$ and $\mathsf{Out}_\alpha(op_i)$ in order to emphasize reference to execution α.

An execution α induces a partial order $\xrightarrow{\alpha}$ on the set of operations in α as follows. For any two operations $op_{i_1} = [\texttt{Invoke}_{i_1}(a_1), \texttt{Response}_{i_1}(b_1)]$ and $op_{i_2} = [\texttt{Invoke}_{i_2}(a_2), \texttt{Response}_{i_2}(b_2)]$ at processes p_{i_1} and p_{i_2}, respectively, say that op_{i_1} *precedes* op_{i_2} *in execution* α, denoted $op_{i_1} \xrightarrow{\alpha} op_{i_2}$, if the response $\texttt{Response}_{i_1}(b_1)$ precedes the invocation $\texttt{Invoke}_{i_2}(a_2)$. In particular, execution α induces, for each process p_i a total order $\xrightarrow{\alpha}_i$ on the set of operations at p_i in α as follows: For any two operations $op_i^{(1)}$ and $op_i^{(2)}$, $op_i^{(1)} \xrightarrow{\alpha}_i op_i^{(2)}$ if and only if $op_i^{(1)} \xrightarrow{\alpha} op_i^{(2)}$.

If, in execution α, operation op_{i_1} does not precede operation op_{i_2}, then we write $op_{i_1} \overset{\alpha}{\not\longrightarrow} op_{i_2}$. If simultaneously $op_{i_1} \overset{\alpha}{\not\longrightarrow} op_{i_2}$ and $op_{i_2} \overset{\alpha}{\not\longrightarrow} op_{i_1}$, then we say that op_{i_1} and op_{i_2} are *parallel* in execution α, denoted as $op_{i_1} \parallel_\alpha op_{i_2}$.

For any execution α of system **P**, a *serialization* $S(\alpha)$ [5] of execution α is a sequence whose elements are the operations of α, and each operation of α appears exactly once in

$S(\alpha)$. Thus, a serialization $S(\alpha)$ is a total order $\xrightarrow{S(\alpha)}$ on the set of operations in α. Notice that there may be, in general, many possible serializations of the execution α. Say that a serialization $S(\alpha)$ is *valid for the monotone group* $\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \preccurlyeq \rangle$ if the following two conditions hold:

1. *Valid Start:* If $op_i = [\texttt{Invoke}_i(a), \texttt{Response}_i(b)]$ is the first operation in $S(\alpha)$, then $b = e$.
2. *Valid Composition:* For any pair of operations $op^{(1)}_{i_1} = [\texttt{Invoke}_{i_1}(a_1), \texttt{Response}_{i_1}(b_1)]$ and $op^{(2)}_{i_2} = [\texttt{Invoke}_{i_2}(a_2), \texttt{Response}_{i_2}(b_2)]$ that are consecutive in $S(\alpha)$, $b_2 = b_1 \oplus a_1$.

Sometimes we shall simply refer to a valid serialization, and avoid explicit reference to the monotone group when such is clear from context.

Say that *system* **P** *implements the monotone group* $\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \preccurlyeq \rangle$ if every execution α of **P** has a serialization that is valid for the monotone group. *Monotone* RMW operations are those associated in the natural way with monotone groups. Say that system **P** implements a (monotone) operation whenever it implements the associated monotone group.

We continue to state and prove the *Unique Serialization Lemma*.

Lemma 3.1 (*Unique Serialization Lemma*). *Assume that system* **P** *implements the monotone group* $\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \preccurlyeq \rangle$. *Then, for any execution* α *of* **P**, *there is a unique valid serialization* $S(\alpha)$.

Proof. Assume, by way of contradiction, that there are two distinct valid serializations $S^{(1)}(\alpha) = op^{(1.1)}, op^{(1.2)}, op^{(1.3)}, \ldots$ and $S^{(2)}(\alpha) = op^{(2.1)}, op^{(2.2)}, op^{(2.3)}, \ldots$ of execution α. Since $S^{(1)}(\alpha)$ and $S^{(2)}(\alpha)$ are distinct, there exists a *least* index $k \geqslant 1$ such that $op^{(1.k)}$ is different from $op^{(2,k)}$. Assume, without loss of generality, that $op^{(1.k)}$ appears at position $l > k$ in the serialization $S^{(2)}(\alpha)$; that is, $op^{(1.k)} = op^{(2.l)}$, so that, in particular, $\mathsf{Out}\left(op^{(1.k)}\right) = \mathsf{Out}\left(op^{(2.l)}\right)$. Notice finally that for each $i < k$,$op^{(1.i)} = op^{(2.i)}$.

We proceed by case analysis on the possible values of k.

1. Assume first that $k = 1$. Since $S^{(1)}(\alpha)$ is a valid serialization of α and $k = 1$, the *Valid Start* condition implies that $\mathsf{Out}(op^{(1.k)}) = e$. Since $S^{(2)}(\alpha)$ is a valid serialization of α and $l > k = 1$, the *Valid Composition* condition implies that

$$\mathsf{Out}(op^{(2.l)}) = \mathsf{Out}(op^{(2.l-1)}) \oplus \mathsf{In}(op^{(2.l-1)}).$$

The *Monotonicity under Composition* property implies that $\mathsf{Out}(op^{(2.l-1)}) \oplus \mathsf{In}(op^{(2.l-1)}) \succ \mathsf{In}(op^{(2.l-1)})$. Since $\mathsf{In}(op^{(2.l-1)}) \in \mathbb{M}$, the *Identity Lower Bound* property implies that $\mathsf{In}(op^{(2.l-1)}) \succ e$. It follows that $\mathsf{Out}(op^{(2.l)}) \succ e$. A contradiction.

2. Assume now that $k > 1$. Since $S^{(1)}(\alpha)$ is a valid serialization of α and $k > 1$, the *Valid Composition* property implies that

$$\mathsf{Out}(op^{(1.k)}) = \mathsf{Out}(op^{(1.k-1)}) \oplus \mathsf{In}(op^{(1.k-1)}).$$

Since $S^{(2)}(\alpha)$ is a valid serialization of α and $l > k > 1$, the *Valid Composition* property implies that

$$\begin{aligned}\mathsf{Out}(op^{(2.l)}) = {}&\mathsf{Out}(op^{(2.k-1)}) \oplus \mathsf{In}(op^{(2.k-1)}) \oplus \cdots \oplus \mathsf{In}(op^{(2.l-2)})\\ &\oplus \mathsf{In}(op^{(2.l-1)}).\end{aligned}$$

Since $\mathsf{Out}(op^{(1.k)}) = \mathsf{Out}(op^{(2.l)})$, it follows that

$$\begin{aligned}&\mathsf{Out}(op^{(1.k-1)}) \oplus \mathsf{In}(op^{(1.k-1)})\\ &\quad = \mathsf{Out}(op^{(2.k-1)}) \oplus \mathsf{In}(op^{(2.k-1)}) \oplus \cdots \oplus \mathsf{In}(op^{(2.l-2)}) \oplus \mathsf{In}(op^{(2.l-1)}).\end{aligned}$$

Since $\mathsf{Out}(op^{(1.k-1)}) = \mathsf{Out}(op^{(2.k-1)})$ and $\mathsf{In}(op^{(1.k-1)}) = \mathsf{In}(op^{(2.k-1)})$, it follows that

$$\begin{aligned}&\mathsf{Out}(op^{(2.k-1)}) \oplus \mathsf{In}(op^{(2.k-1)})\\ &\quad = \mathsf{Out}(op^{(2.k-1)}) \oplus \mathsf{In}(op^{(2.k-1)}) \oplus \cdots \oplus \mathsf{In}(op^{(2.l-2)}) \oplus \mathsf{In}(op^{(2.l-1)}).\end{aligned}$$

By the *Cancellation Law*, it follows that

$$e = \mathsf{In}(op^{(2.k)}) \oplus \cdots \oplus \mathsf{In}(op^{(2.l-2)}) \oplus \mathsf{In}(op^{(2.l-1)}).$$

The *Monotonicity under Composition* property implies that

$$\mathsf{In}(op^{(2.k)}) \oplus \cdots \oplus \mathsf{In}(op^{(2.l-2)}) \oplus \mathsf{In}(op^{(2.l-1)}) \succ \mathsf{In}(op^{(2.k)}).$$

Since $\mathsf{In}(op^{(2.k)}) \in \mathbb{M}$, the *Identity Lower Bound* property implies that $\mathsf{In}(op^{(2.k)}) \succ e$. It follows that

$$\mathsf{In}(op^{(2.k)}) \oplus \cdots \oplus \mathsf{In}(op^{(2.l-2)}) \oplus \mathsf{In}(op^{(2.l-1)}) \succ e.$$

A contradiction.

Since we obtained a contradiction in all possible cases, the proof is now complete. □

We remark that the proof of Lemma 3.1 relied heavily on the required properties for a monotone group, namely the *Monotonicity under Composition* and *Identity Lower Bound* properties. Since these properties do not necessarily hold for a *general* group, the same follows for the *Unique Serialization Lemma*. We conclude this section with an immediate consequence of the *Valid Start* and *Valid Composition* conditions assumed in the definition of implementation of a monotone group.

Property 3.2. *Assume that system* **P** *implements the monotone group* $\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \preccurlyeq\rangle$. *Then, for any operation op in an execution* α *of* **P**,

$$\mathsf{Out}_\alpha(op) = \biguplus_{|\{op' \mid op' \xrightarrow{S(\alpha)} op\}|} \left\{ \mathsf{In}_\alpha\left(op'\right) \mid op' \xrightarrow{S(\alpha)} op \right\}.$$

3.2. Linearizability

Our definitions refer to a distributed system **P** implementing a monotone group $\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \preccurlyeq\rangle$, and, in particular, to any arbitrary execution α of it and its (unique) valid serialization $S(\alpha)$.

Say that execution α is *linearizable* [10] if the serialization $S(\alpha)$ extends $\xrightarrow{\alpha}$; that is, for any pair of operations $op^{(1)}$ and $op^{(2)}$ such that $op^{(1)} \xrightarrow{\alpha} op^{(2)}$, $op^{(1)} \xrightarrow{S(\alpha)} op^{(2)}$. The *Valid Composition* condition implies that for any two operations $op^{(1)}$ and $op^{(2)}$ such that $op^{(1)} \xrightarrow{S(\alpha)} op^{(2)}$, $\mathsf{Out}_\alpha(op^{(1)}) \prec \mathsf{Out}_\alpha(op^{(2)})$. Thus, it follows that for any pair of operations $op^{(1)}$ and $op^{(2)}$ such that $op^{(1)} \xrightarrow{\alpha} op^{(2)}$, $\mathsf{Out}_\alpha(op^{(1)}) \prec \mathsf{Out}_\alpha(op^{(2)})$.

Say that operation $op^{(1)}$ in execution α is *non-linearizable in execution* α if there is another operation $op^{(2)}$ in execution α such that $op^{(2)} \xrightarrow{\alpha} op^{(1)}$ while $op^{(2)} \xrightarrow{S(\alpha)} op^{(1)}$. Say that operation op in execution α is *linearizable in execution* α if it is not non-linearizable in execution α. It follows that execution α is linearizable if every operation in execution α is linearizable in it. Finally, we say that system **P** is *linearizable* if all its executions are.

4. Switching networks

In this section, we present a framework for switching networks. Some of our definitions are common with some from [6, Section 2] and [7, Section 2], while most of them refine and extend corresponding ones there. Some basic definitions are articulated in Section 4.1. Processes, tokens, switches and wires are described in Section 4.2. Section 4.3 defines states, configurations and executions. The outputs of switching networks are described in Section 4.4. Section 4.5 introduces some contention measures for switching networks.

4.1. Basic definitions

A *switching network* [6], like a *counting network* [2], is a directed (acyclic) graph in which the nodes are simple computing elements called *switches*, and the edges are called *wires*.

More specifically, an $(f_{\text{in}}, f_{\text{out}})$*-switch*, or *switch* for short, is a routing element with f_{in} *input wires*, f_{out} *output wires*, and an *internal state*; f_{in} and f_{out} are called the switch's *fan-in* and *fan-out*, respectively. A switch's internal state is a collection of variables, possibly with initial values. In the *initial* state of switch, all of its variables are set to their initial values. The number of internal states of a switch may be either finite or infinite, giving rise to a *finite-state* or *infinite-state* switch, respectively. In either case, a switch changes its internal state according to its *transition function*.

A *finite-state switching network* is a switching network made up from finite-state switches; an *infinite-state switching network* is a switching network made up from infinite-state switches.

A $(w_{\text{in}}, w_{\text{out}})$*-switching network* $\mathcal{N}$ has w_{in} *input wires* and w_{out} *output wires*, and it is formed by connecting together switches; thus, we connect output wires of switches to input wires of other switches. Some switches have input wires (resp., output wires) not connected to other switches in the network, and these wires are the w_{in} input wires (resp., w_{out} output wires) of the switching network $\mathcal{N}$.

The *size* $S(\mathcal{N})$ of a switching network $\mathcal{N}$ is the total number of its switches. A network $\mathcal{N}$ is *finite-size* if $s(\mathcal{N}) < \infty$; else, it is *infinite-size*. The *depth* $d(b)$ of a switch b in a switching network $\mathcal{N}$ is defined to be 0 if one of its input wires is an input wire of the network, and $\max_j d(b_j)+1$, where the maximum is taken over all switches b_j with output wires connected to input wires of switch b. The *depth* $d(\mathcal{N})$ of the network $\mathcal{N}$ is defined as the maximum depth of any of its switches. The switching network $\mathcal{N}$ can naturally be divided into $d(\mathcal{N})$ *layers*, so that layer ℓ contains all switches of depth ℓ, where $0 \leqslant \ell \leqslant d(\mathcal{N})$. A *path* in a switching network is a sequence of switches each (other than the last) connected to the next.

4.2. Processes, tokens and switches

We assume a collection of asynchronous, non-failing processes that access a switching network by shepherding *tokens* through it. A switching network may be accessed by many tokens simultaneously, which traverse the network asynchronously; however, each process has at most one token sheperded through the network at each time. The *concurrency* of a switching network is the maximum number of processes (and, therefore, tokens as well) allowed to access the network simultaneously.

Unlike counting networks [2], each token has a *state* (a collection of variables) which is allowed to change as the token traverses the network according to its *transition function*. The state of a token includes its *input value*.

A token enters the switching network on one of the network's w_{in} input wires. Then, the token is instantaneously forwarded to the switch to which the wire belongs; the switch then routes the token to one of its output wires from which the token enters the next switch in the network, and so on. Both the switch's and the token's states change. The token continues traversing the network in the same fashion until it reaches one of the w_{out} output wires of the network. At that point, the token exits the network and returns a value to the process that owns it.

In more detail, when a token arrives on an input wire of a switch, the following events occur in a single, *atomic* (indivisible) step:

> The switch removes the token from the input wire and it changes state; the token changes state and it is routed to an output wire of the switch.

For example, an $(f_{\text{in}}, f_{\text{out}})$ *balancer* is a finite-state switch with fan-in f_{in} and fan-out f_{out}. The kth token to arrive on any of its input wires is routed to the output wire $f_{\text{out}} \bmod k$. Thus, the state of an $(f_{\text{in}}, f_{\text{out}})$ balancer encapsulates the number of tokens that have traversed the switch modulo its fan-out f_{out}. The state of a token traversing an $(f_{\text{in}}, f_{\text{out}})$-balancer is not affected. Such balancers have been used to construct counting networks (see, e.g., [2,9]).

4.3. States, configurations and executions

For each $(f_{\text{in}}, f_{\text{out}})$-switch, denote by x_i, $0 \leqslant i \leqslant f_{\text{in}}-1$, the number of tokens that have entered the switch on input wire i; similarly, denote by y_j the number of tokens that have exited the switch on output wire j.

A switch's *state* includes both its internal state and the collections of tokens on its input and output wires. A switch is in a *quiescent* state if there are no tokens currently traversing the switch; thus, in a quiescent state, the number of tokens that arrived on the input wires of the switch have exited the switch on its output wires, or $\sum_{i=1}^{f_{\text{in}}} x_i = \sum_{j=1}^{f_{\text{out}}} y_j$.

A switch satisfies the following two conditions:

1. *Safety condition:* In any state, $\sum_{i=1}^{f_{\text{in}}} x_i \geqslant \sum_{j=1}^{f_{\text{out}}} y_j$. thus, a switch never creates tokens spontaneously.
2. *Liveness condition:* Starting from any state, a switch eventually reaches a quiescent state.

An *internal configuration* of a switching network is a collection of the internal states of its switches. Consider a finite-state switching network $\mathcal{N}$ with (finite) switches having S internal states each. Then, clearly, the number of internal configurations of the network $\mathcal{N}$ is finite and equal to $S^{s(\mathcal{N})}$. Note that the number of internal configurations of an infinite-state switching network is no longer finite.

A *configuration* of a switching network is the collection of the states of its switches; thus, the configuration of a switching network includes the states of all tokens currently traversing the network as well. A configuration of a switching network is *quiescent* if all of its switches are in a quiescent state. The safety and liveness properties for switches immediately imply corresponding safety and liveness properties for a switching network.

For any token t and switch s, we denote by $\tau = \langle t, s \rangle$ the *state transition* in which the token t passes (in a single atomic step) from an input wire to an output wire of switch s; thus, in a state transition the state of a switch (including the states of tokens on its input and output wires) changes according to the transition function of the switch (and the transition functions of the tokens on its input and output wires). Although state transitions can occur concurrently, it is convenient to treat them using a model of interleaving semantics.

An *execution* of a switching network is a finite or infinite sequence $\alpha = Q_0, \tau_1, Q_1, \tau_2, Q_2, \ldots$, of alternating configurations and switch transitions such that:

1. Q_0 is the *initial* configuration, in which there are no tokens on input wires of switches except for at least one token on input wires of the network, and all switches are in their initial internal states.
2. For each triple $\langle Q_i, \tau_{i+1}, Q_{i+1} \rangle$, where $i \geqslant 0$, the switch transition τ_{i+1} carries the configuration Q_i to the configuration Q_{i+1}.

A finite execution ends with a configuration. A finite execution is *complete* if it results to a quiescent configuration. An execution α is *sequential* if for any two transitions $\tau_i = \langle t, s_i \rangle$ and $\tau_j = \langle t, s_j \rangle$ that involve the same token t, all transitions (if any) between them also involve that token. Lightly speaking, tokens traverse the network one completely after the other in a sequential execution.

An *execution suffix* of a switching network is a suffix of some execution of the network that starts with a configuration. The definition of sequential executions can be extended to *sequential execution suffixes* in the natural way. So again, tokens traverse the network one completely after the other in a sequential execution suffix.

An *execution fragment* of a switching network is a finite (contiguous) subsequence of some execution of the network that starts and ends with a configuration. A *pump* of a switching network is an execution fragment of it that starts and ends with the same quiescent configuration. The *concatenation* $\alpha_1 \cdot \alpha_2$ of two execution fragments α_1 and α_2 is defined when α_2 follows α_1 in the same execution of the network; thus, the end configuration of α_1 is the start configuration of α_2. The concatenation is also an execution fragment; thus, it does not repeat the common configuration of the two original execution fragments.

For an execution of a switching network, we say that concurrency is *bounded* if the number of concurrent processes accessing the network in the execution is bounded. In an (infinite) execution, we say that concurrency is *unbounded* if the number of concurrent processes accessing the network in the execution is unbounded (either finite or infinite).

4.4. Outputs

The input and output values of token t in execution α will be denoted as $\mathsf{In}_{\alpha}(t)$ and $\mathsf{Out}_{\alpha}(t)$, respectively.

For finite-state switching networks, we include an additional component on the output wires of the switching network, namely the *output registers*. More specifically, there is an output register associated with each output wire of the switching network. However, unlike finite switches, each output register has an infinite number of states. Denote $or(\mathcal{N})$ the number of output registers in a finite-state switching network $\mathcal{N}$.

The *output value* for a token in a finite-state switching network is computed on the output register residing on the network's output wire from which the token exits. When a token arrives on an output register. the following events occur in a single, *atomic* (indivisible) step:

1. The token computes its output value according to the output register's state.
2. The state of the output register changes according to its previous state and the state of the token (which includes its input value).

Note that the input value of a token does not affect its output value, but it may as well affect the output values of tokens that will later access the same output register.

We remark that finite-state switching networks correspond more closely to traditional counting networks [2], where a token fetching the counter's value and incrementing the counter by one obtains the value from the register attached to the output wire it will exit from. We also remark that output registers are *necessary* for this kind of switching networks, since they provide an infinite number of different output values to tokens, while finite switches, used only for routing, are unable to do so.

For infinite-state switching networks, there are no attached output registers and the *output value* of a token is determined according to the state of the token when it exits the network.

4.5. Contention measures

In a switching network, contention represents the extent to which concurrent processes access the same switch or output register simultaneously. We use two complexity-theoretic measures to model contention in switching networks, namely *register bottleneck* and *switch bottleneck*, which are introduced here for the first time.

The definition of register bottleneck applies only to finite-state switching networks.

Definition 4.1 (*Register bottleneck*). The *register bottleneck* of a finite-state switching network $\mathcal{N}$ is the *minimum* number of output registers, where the minimum is taken over all infinite executions of the network, that are accessed by tokens in some infinite suffix of an infinite execution of the network.

On the account of register bottleneck, a switching network is *low-contention* if its register bottleneck is sufficiently large. A register bottleneck of 1 is the *worst* possible register bottleneck, since it implies the existence of some execution of the network in which as many tokens as processes participating in the execution will eventually accumulate in front of the same output register, which thus becomes a “hot-spot”. Note that register bottleneck is a trivial lower bound on the number of output registers of a finite-state switching network. We prove:

Lemma 4.1. *Assume that the register bottleneck of $\mathcal{N}$ is at least* 2. *Then, in any pump of $\mathcal{N}$, there exist at least two distinct tokens that access two different output registers*.

Proof. Assume, by way of contradiction, that there is a pump ϕ of $\mathcal{N}$ in which all tokens access the same output register. Clearly, the infinite sequence $\phi \cdot \phi \cdot \ldots$ of pumps is an infinite suffix of an infinite execution of $\mathcal{N}$ in which all tokens access the same output register. It follows that the register bottleneck of $\mathcal{N}$ is 1. A contradiction. $\square$

The definition of switch bottleneck will be useful for infinite-state switching networks.

Definition 4.2 (*Switch bottleneck*). The *switch bottleneck* of a switching network $\mathcal{N}$ is the *minimum* number of switches, where the minimum is taken over all infinite executions of the network, that are accessed by an infinite sequence of tokens exiting a switch connected to them that has been accessed by an infinite number of tokens itself.

On the account of switch bottleneck, a switching network is *low-contention* if its switch bottleneck is sufficiently large. A switch bottleneck of 1 is the *worst* possible switch bottleneck since it implies the existence of some infinite execution of the network in which some switch is accessed by an infinite number of tokens and it outputs a finite number of tokens on all but one of its output wires. Intuitively, such a switch does not effectively “balance” the infinite stream of tokens that access it, but it emits almost all of them (except for a finite number) to the same switch in the next layer; this last switch will eventually become a “hot-spot”.

Clearly, in the special case where switches are balancers which “balance” their input tokens, the switch bottleneck is the least (over all balancers) number of output wires of a balancer, which (usually) exceeds 1. Thus, the requirement that switch bottleneck be high can also be seen as a generalization of the balancing property from balancers to general switches.

Note that switch bottleneck is a trivial lower bound on the number of switches in any layer (other than layer 1) of an infinite-switch network. In our later proofs, we will also assume that this is also a lower bound for layer 1.

Consider a switching network with a certain switch bottleneck. Consider now what happens when some tokens have been permanently “halted” in front of some switches of the network in some infinite sequence; this resulting sequence is not necessarily an execution since it fails to guarantee liveness. Recall, however, that a switch operates locally: it changes its state according to its state and the states of tokens that traverse it, and independently of the operation of other tokens and switches in the network. This implies that the switch bottleneck of the network is maintained also for such sequences. (This observation will be used in the proof of Theorem 6.2.)

4.6. Switching networks implementing monotone groups

A switching network $\mathcal{N}$ can be used to implement a monotone group $\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \preccurlyeq \rangle$ as follows:

- Token t issued by process p_i corresponds to an operation $op_i = [\mathtt{Invoke}_i(a), \mathtt{Response}_i(b)]$ invoked by process p_i, where $a \in \mathbb{M}$ and $b \in \mathbb{M} \cup \{e\}$. We say that a is the *input value* or *type* of the token t, and b is the *output value* of the token t. The input value of the token is part of the token's (initial) state.
- For any execution α, the invocation of operation op_i corresponds to the first transition $\tau_i = \langle t_i, s_i \rangle$ in execution α, where $t_i = t$ and s_i is an input switch of the network; this transition occurs when the token enters the network. The response of operation op corresponds to the latest transition $\tau_j = \langle t_j, s_j \rangle$ in execution α, where $t_j = t$ and s_j is an output switch of the network; this transition occurs when the token exits the network.
- When token t exits the network, it carries encapsulated in its state the output value b that operation op_i is returned.

It is now straightforward to formally define when the *switching network $\mathcal{N}$ implements the monotone group* $\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \preccurlyeq \rangle$.

4.7. The covering technique

In some of our impossibility proofs, we will use a variant of the *variable covering* technique originally introduced by Burns and Lynch [3] for proving lower bounds on the number of read/write registers needed to solve (deadlock-free) mutual exclusion. Intuitively, a token covers a switch if it is about to access the switch. We omit the formal definition here, which can be immediately extended to tokens covering output registers as well.

5. The Monotone Linearizability Lemma

Throughout this section, we refer to a distributed system **P** implementing a monotone group $\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \preccurlyeq \rangle$. The main contribution of the section is to state and prove the *Monotone Linearizability Lemma*, which establishes ordering constraints of linearizability on the system **P**. Recall that, by Lemma 2.5, the monotone group $\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \preccurlyeq \rangle$ is n-wise independent for any integer $n \geqslant 2$. So, there are n distinct elements $a_1, a_2, \ldots, a_n \in \mathbb{M}$, with $a_1, a_2, \ldots, a_n \neq e$, which are n-wise independent over $\langle \mathbb{\Gamma}, \oplus \rangle$. The proof of the *Monotone Linearizability Lemma* amounts to establishing a contradiction to n-wise independence for a hypothetical *non-linearizable* execution, in which the types of the RMW operations issued by the processes are $a_1, a_2, \ldots, a_n$. We are now ready to state and prove the *Monotone Linearizability Lemma*.

Proposition 5.1 (*Monotone Linearizability Lemma*). *Consider any execution α of system* **P** *in which each process p_i issues only operations of type a_i, where $1 \leqslant i \leqslant n$. Then, α is linearizable.*

Proof. We start with an informal outline of our proof. We will proceed by contradiction. We will consider the earliest non-linearizable operation op_k (at process p_k) in α and the latest operation op_l that precedes it. We will use these two operations to construct two executions γ_1 and γ_2 that are indistinguishable to process p_l with respect to operation op_l. This indistinguishability implies that op_l receives the same output in these two executions. The contradiction will follow from the comparison of the two identical outputs, where we use simple algebraic properties of (monotone) groups in order to contradict the assumed n-wise independence. We now continue with the details of the formal proof.

Assume, by way of contradiction, that α is *not* linearizable. So, there is at least one operation that is non-linearizable in execution α. Consider the *earliest* such operation op_k (occurring at process p_k), and let op_l be the *latest* operation (occurring at process p_l) that precedes op_k in α. So, $op_l \xrightarrow{\alpha} op_k$ while $op_k \xrightarrow{S(\alpha)} op_l$, where $S(\alpha)$ is the (unique) valid serialization of α.

In our proof, we will use the operations op_k and op_l in order to define and treat two finite prefixes of execution α:

- the finite prefix β_1 of execution α that ends with the response for operation op_k, and
- the finite prefix β_2 of execution α that ends with the response for operation op_l.

Clearly, β_2 is a prefix of β_1 as well. We first treat separately each of the two prefixes β_1 and β_2 and a corresponding extension of it; we then treat them together.

Properties of the prefix β_1 and its extension γ_1: Consider a finite execution γ_1, which is an extension of β_1 that includes no additional invocations by processes; so, γ_1 is extended to only include responses to invocations that are pending in β_1.

Since β_1 is a prefix of both α and γ_1, it follows that all operations whose responses are included in β_1 (or, in other words, they are not preceded in either α or γ_1 by the response for op_k) have identical outputs in α and γ_1. In particular, $\mathsf{Out}_\alpha(op_l) = \mathsf{Out}_{\gamma_1}(op_l)$ and $\mathsf{Out}_\alpha(op_k) = \mathsf{Out}_{\gamma_1}(op_k)$. Take now the (unique) valid serialization $S(\gamma_1)$ of γ_1.

Since $op_k \xrightarrow{S(\alpha)} op_l$ the *Valid Composition* condition (for $S(\alpha)$) implies that $\mathsf{Out}_\alpha(op_k) \prec \mathsf{Out}_\alpha(op_l)$. Since $\mathsf{Out}_\alpha(op_k) = \mathsf{Out}_{\gamma_1}(op_k)$ and $\mathsf{Out}_\alpha(op_l) = \mathsf{Out}_{\gamma_1}(op_l)$, it follows that $\mathsf{Out}_{\gamma_1}(op_k) \prec \mathsf{Out}_{\gamma_1}(op_l)$. The *Valid Composition* condition (for $S(\gamma_1)$) implies now that $op_k \xrightarrow{S(\gamma_1)} op_l$.

For each process p_i, where $1 \leqslant i \leqslant n$, denote $\mu_i^{(1)}$ the number of operations at p_i that precede op_l in the serialization $S(\gamma_1)$. Assume that:

- $\mu_{i,a}^{(1)}$ of those $\mu_i^{(1)}$ operations have their responses followed in γ_1 by that for op_l;
- the rest $\mu_{i,b}^{(1)}$ of them have their responses preceded in γ_1 by that for op_l.

So, $\mu_i^{(1)} = \mu_{i,a}^{(1)} + \mu_{i,b}^{(1)}$. We next prove a simple property.

Property 5.2. *For each process p_i, where $1 \leqslant i \leqslant n$, $\mu_{i,b}^{(1)} \leqslant 2$.*

Proof. Consider the *earliest* (if any) operation op at process p_i such that $op \xrightarrow{S(\gamma_1)} op_l$, while the response for op follows the one for op_l in γ_1. We proceed by case analysis on the order of the responses for op and op_k in γ_1.

1. Assume first that the response for op follows the one for op_k in γ_1. Since γ_1 includes no invocations following the response for op_k, it follows that there is no other operation at p_i following op, so that $\mu_{i,b}^{(1)} \leqslant 1$ in this case.

2. Assume now that the response for *op* precedes the one for op_k in γ_1. Consider any other operation op' at p_i that follows *op* in γ_1, while still $op' \xrightarrow{S(\gamma_1)} op_l$. We will prove that there is at most one such additional operation.
   - By construction of γ_1, the invocation for op' precedes the response for op_k in γ_1.
   - Assume, by way of contradiction, that the response for op' precedes the response for op_k in γ_1. Thus, op' is included in the prefix β_1. Since β_1 is a prefix of α, it follows that the response for op' precedes the response for op_k in α as well. This implies that $\mathsf{Out}_{\gamma_1}\left(op'\right) = \mathsf{Out}_{\alpha}\left(op'\right)$. Since $op' \xrightarrow{S(\gamma_1)} op_l$, the *Valid Composition* condition (for $S(\gamma_1)$) implies that $\mathsf{Out}_{\gamma_1}\left(op'\right) \prec \mathsf{Out}_{\gamma_1}\left(op_l\right)$. Since $\mathsf{Out}_{\gamma_1}\left(op_l\right) = \mathsf{Out}_{\alpha}\left(op_l\right)$, it follows that $\mathsf{Out}_{\alpha}\left(op'\right) \prec \mathsf{Out}_{\alpha}\left(op_l\right)$. The *Valid Composition* condition (for $S(\alpha)$) implies now that $op' \xrightarrow{S(\alpha)} op_l$.

     Since the response for *op* follows the response for op_l in γ_1, while op' follows *op* in γ_1, it follows that $op_l \xrightarrow{\gamma_1} op'$. Since op' is included in the prefix β_1 of γ_1, which is also a prefix of α, this implies that $op_l \xrightarrow{\alpha} op'$ as well. It follows that op' is a non-linearizable operation in α. Since the response for op' precedes the response for op_l in α, it follows that op' is an earlier than op_l, non-linearizable operation in α. A contradiction.

     It follows that the response for op' follows the response for op_k in γ_1.

   Since γ_1 includes no invocations following the response for op_k, it follows that there is no other operation at p_i following op' in γ_1, so that $\mu_{i,b}^{(1)} \leqslant 2$ in this case.

Thus, in all cases, $\mu_{i,b}^{(1)} \leqslant 2$, as needed. □

Since $op_k \xrightarrow{S(\gamma_1)} op_l$ while the response for op_l precedes the response for op_k in γ_1, a slight strengthening of Property 5.2 for the particular case of process p_k is now immediate:

Property 5.3. $1 \leqslant \mu_{k,b}^{(1)} \leqslant 2$

By Property 3.2, $\mathsf{Out}_{\gamma_1}\left(op_l\right)$ is a composite expression involving for each process p_i, $1 \leqslant i \leqslant n$, $\mu_i^{(1)}$ contributions of a_i. By the *Commutativity* property, these *n* types of contributions can be separated from each other in the composite expression, so that

$$\mathsf{Out}_{\gamma_1}\left(op_l\right) = \biguplus_{i=1}^{n} \bigoplus_{\mu_i^{(1)}} a_i.$$

Properties of the prefix β_2 and its extension γ_2: Consider a finite execution γ_2, which is an extension of β_2 that includes no additional invocations by processes; so, γ_2 is an extension that only includes responses to invocations that are pending in β_2 (in addition to the responses included in β_2).

Since β_2 is a prefix of both α and γ_2, it follows that all operations whose responses are included in β_2 (hence, they are not preceded in either α or γ_2 by the response for op_l) have identical outputs in α and γ_2. In particular, $\mathsf{Out}_{\alpha}\left(op_l\right) = \mathsf{Out}_{\gamma_2}\left(op_l\right)$. Take now the (unique) valid serialization $S(\gamma_2)$ of γ_2.

For each process p_i, where $1 \leqslant i \leqslant n$, denote $\mu_i^{(2)}$ the number of operations at p_i that precede op_l in the serialization $S(\gamma_2)$. Assume that:

- $\mu_{i,a}^{(2)}$ of those $\mu_i^{(2)}$ operations have their responses not preceded by that for op_l in γ_2;
- the rest $\mu_{i,b}^{(2)}$ of them have their responses preceded in γ_2 by that for op_l.

So, $\mu_i^{(2)} = \mu_{i,a}^{(2)} + \mu_{i,b}^{(2)}$. We continue to prove a simple property:

Property 5.4. *For each process p_i, where $1 \leqslant i \leqslant n$, $\mu_{i,b}^{(2)} \leqslant 1$.*

Proof. Consider the *earliest* (if any) operation *op* at process p_i such that $op \xrightarrow{S(\gamma_2)} op_l$, while the response for *op* follows the one for op_l in γ_2. Since γ_2 includes no invocations following the response for op_l, it follows that there is no other operation at p_i following *op* in γ_2, so that $\mu'_{i,b} \leqslant 1$, as needed. □

By Property 3.2, $\mathsf{Out}_{\gamma_2}(op_l)$ is a composite expression involving for each process p_i, $1 \leqslant i \leqslant n$, $\mu_i^{(2)} = \mu_{i,a}^{(2)} + \mu_{i,b}^{(2)}$ contributions of a_i. By the *Commutativity* property, these *n* types of contributions can be separated from each other in the composite expression, so that

$$\mathsf{Out}_{\gamma_2}(op_l) = \biguplus_{i=1}^{n} \bigoplus_{\mu_i^{(2)}} a_i .$$

Joint properties of the prefixes β_1 and β_2 and their extensions γ_1 and γ_2: Since $\mathsf{Out}_\alpha(op_l) = \mathsf{Out}_{\gamma_1}(op_l)$ and $\mathsf{Out}_\alpha(op_l) = \mathsf{Out}_{\gamma_2}(op_l)$, it follows that $\mathsf{Out}_{\gamma_1}(op_l) = \mathsf{Out}_{\gamma_2}(op_l)$.

We continue to prove two simple properties of the prefixes β_1 and β_2, and their extensions γ_1 and γ_2. The first property relates $\mu_{i,a}^{(1)}$ and $\mu_{i,a}^{(2)}$, while the second one relates $\mu_{i,b}^{(1)}$ and $\mu_{i,b}^{(2)}$. We start with the first.

Property 5.5. *For each process p_i, where $1 \leqslant i \leqslant n$, $\mu_{i,a}^{(1)} = \mu_{i,a}^{(2)}$.*

Proof. We will prove that both $\mu_{i,a}^{(1)} \leqslant \mu_{i,a}^{(2)}$ and $\mu_{i,a}^{(2)} \leqslant \mu_{i,a}^{(1)}$.

1. To prove that $\mu_{i,a}^{(1)} \leqslant \mu_{i,a}^{(2)}$, consider any operation *op* at process p_i such that $op \xrightarrow{S(\gamma_1)} op_l$, while the response for *op* precedes the response for op_l in γ_1. So, *op* is included in prefix β_2.
    - Since β_2 is a prefix of γ_2, and it ends with the response for operation op_l, it follows that the response for *op* precedes the response for op_l in γ_2 as well.
    - Since β_2 is a prefix of both γ_1 and γ_2, it follows that $\mathsf{Out}_{\gamma_1}(op) = \mathsf{Out}_{\gamma_2}(op)$. Since $op \xrightarrow{S(\gamma_1} op_l$, the *Valid Composition* condition for $S(\gamma_1)$ implies that $\mathsf{Out}_{\gamma_1}(op) \prec \mathsf{Out}_{\gamma_1}(op_l)$. Since $\mathsf{Out}_{\gamma_1}(op_l) = \mathsf{Out}_{\gamma_2}(op_l)$, it follows that $\mathsf{Out}_{\gamma_2}(op_l) = \mathsf{Out}_{\gamma_2}(op_l)$. Thus, the *Valid Composition* condition for $S(\gamma_2)$ implies that $op \xrightarrow{S(\gamma_2)} op_l$.

    It follows that $\mu_{i,a}^{(1)} \leqslant \mu_{i,a}^{(2)}$.

2. To prove that, $\mu^{(1)}_{i,a} \leqslant \mu^{(2)}_{i,a}$, consider any operation *op* at process p_i such that $op \xrightarrow{S(\gamma_2)} op_l$, while the response for *op* precedes the response for op_l in γ_2. So, *op* is included in prefix β_2.
   - Since β_2 is a prefix of γ_1, and it ends with the response for operation op_l, it follows that the response for *op* precedes the response for op_l in γ_1 as well.
   - Since β_2 is a prefix of both γ_1 and γ_2, it follows that $\mathsf{Out}_{\gamma_1}(op) = \mathsf{Out}_{\gamma_2}(op)$. Since $op \xrightarrow{S(\gamma_2)} op_l$, the *Valid Composition* condition for $S(\gamma_2)$ implies that $\mathsf{Out}_{\gamma_2}(op) \prec \mathsf{Out}_{\gamma_2}(op_l)$. Since $\mathsf{Out}_{\gamma_1}(op_l) = \mathsf{Out}_{\gamma_2}(op_l)$, it follows that $\mathsf{Out}_{\gamma_1}(op_l) = \mathsf{Out}_{\gamma_1}(op_l)$. Thus, the *Valid Composition* condition for $S(\gamma_1)$ implies that $op \xrightarrow{S(\gamma_1)} op_l$.

   It follows that $\mu^{(1)}_{i,a} \leqslant \mu^{(2)}_{i,a}$.

So, in total, $\mu^{(1)}_{i,a} = \mu^{(2)}_{i,a}$, as needed. □

We continue with the second property.

Property 5.6. $\mu^{(1)}_{k,b} - \mu^{(2)}_{k,b} \geqslant 1$.

Proof. Recall from Property 5.3 that $1 \leqslant \mu^{(1)}_{k,b} \leqslant 2$. We proceed by case analysis on $\mu^{(1)}_{k,b}$.

1. Assume first that $\mu^{(1)}_{k,b} = 1$. Since $op_k \xrightarrow{S(\gamma_1)} op_l$ and the response for op_k follows the response for op_l in γ_1, it follows that op_k counts for $\mu^{(1)}_{k,b}$. Since $\mu^{(1)}_{k,b} = 1$, this implies that no operation (in γ_1) other than op_k counts for $\mu^{(1)}_{k,b}$; that is, there is no operation op'_k (other than op_k) at process p_k in γ_1 such that $op'_k \xrightarrow{S(\gamma_1)} op_l$ while the response for op'_k follows the response for op_l in γ_1.

   We will prove that $\mu^{(2)}_{k,b} = 0$ in this case. Assume, by way of contradiction, that $\mu^{(2)}_{k,b} \neq 0$. Thus, there is some operation op'_k in γ_2 such that $op'_k \xrightarrow{S(\gamma_2)} op_l$ while the response for op'_k follows the response for op_l in γ_2. Since γ_2 includes no invocations following the response for op_l, it follows that the invocation for op'_k precedes the response for op_l in γ_2. So, the invocation for op'_k is included in the prefix β_2 of γ_2. Since β_2 is a prefix of both α and γ_1 as well, it follows that op'_k is an operation in each of α and γ_1 as well such that its invocation precedes the response for op_l in each of α and γ_1.

   Since the invocation for op'_k precedes the response for op_l in α (resp., γ_1) and $op_l \xrightarrow{\alpha} op_k$ (resp., $op_l \xrightarrow{\gamma_1} op_k$), it follows that $op'_k \xrightarrow{\alpha} op_k$ (resp., $op'_k \xrightarrow{\gamma_1} op_k$).

   Since the response for op'_k is not included in prefix β_2, it follows that the response for op'_k follows the response for op_l in each of α and γ_1 as well. This implies that $op_l \xrightarrow{S(\gamma_1)} op'_k$. Since $op_l \xrightarrow{S(\gamma_1)} op'_k$, the *Valid Composition* condition for γ_1 implies that $\mathsf{Out}_{\gamma_1}(op_l) \prec \mathsf{Out}_{\gamma_1}(op'_k)$. On the other hand, since $op'_k \xrightarrow{\gamma_1} op_k$, the response for op'_k is included in prefix β_1, which is a prefix of both α and γ_1, so that $\mathsf{Out}_{\alpha}(op'_k) = \mathsf{Out}_{\gamma_1}(op'_k)$. Since also $\mathsf{Out}_{\alpha}(op_l) = \mathsf{Out}_{\gamma_1}(op_l)$, while by assumption, $\mathsf{Out}_{\alpha}(op_k) \prec \mathsf{Out}_{\alpha}(op_l)$, it follows that $\mathsf{Out}_{\alpha}(op_k) \prec \mathsf{Out}_{\alpha}(op'_k)$.

Thus, in total, $\mathsf{Out}_\alpha\left(op_k\right) \prec \mathsf{Out}_\alpha\left(op'_k\right)$, $op'_k \xrightarrow{\alpha} op_k$, and the response for op_k follows the response for op_l in α. So, op_l is *not* the latest operation in α that precedes op_k in α and yet $\mathsf{Out}_\alpha\left(op_k\right) \prec \mathsf{Out}_\alpha\left(op_l\right)$. A contradiction.

The contradiction implies that $\mu_{k,b}^{(2)} = 0$, so that $\mu_{k,b}^{(1)} - \mu_{k,b}^{(2)} = 1$ in this case.

2. Assume now that $\mu_{k,b}^{(1)} = 2$. By Property 5.4, $\mu_{k,b}^{(2)} \leqslant 1$, so that $\mu_{k,b}^{(1)} - \mu_{k,b}^{(2)} \geqslant 1$ in this case.

Thus, in all cases, $\mu_{k,b}^{(1)} - \mu_{k,b}^{(2)} \geqslant 1$, as needed. □

Since $\mathsf{Out}_{\gamma_1}\left(op_l\right) = \mathsf{Out}_{\gamma_2}\left(op_l\right)$, we have that

$$\biguplus_{i=1}^{n} \bigoplus_{\mu_i^{(1)}} a_i = \biguplus_{i=1}^{n} \bigoplus_{\mu_i^{(2)}} a_i .$$

By Property 2.1, it follows that for each process p_i, where $1 \leqslant i \leqslant n$,

$$\bigoplus_{\mu_i^{(1)}} a_i = \bigoplus_{\mu_i^{(1)} - \mu_i^{(2)}} a_i \oplus \bigoplus_{\mu_i^{(2)}} a_i .$$

It follows that

$$\begin{aligned}
\biguplus_{i=1}^{n} \left(\bigoplus_{\mu_i^{(1)}} a_i \right) &= \biguplus_{i=1}^{n} \left(\bigoplus_{\mu_i^{(1)} - \mu_i^{(2)}} a_i \oplus \bigoplus_{\mu_i^{(2)}} a_i \right) \\
&= \left(\bigoplus_{\mu_1^{(1)} - \mu_1^{(2)}} a_1 \oplus \bigoplus_{\mu_1^{(2)}} a_1 \right) \oplus \cdots \oplus \left(\bigoplus_{\mu_n^{(1)} - \mu_n^{(2)}} a_n \oplus \bigoplus_{\mu_n^{(2)}} a_n \right) \\
&\quad \text{(by definition of the summation operator)} \\
&= \left(\bigoplus_{\mu_1^{(1)} - \mu_1^{(2)}} a_1 \oplus \cdots \oplus \bigoplus_{\mu_n^{(1)} - \mu_n^{(2)}} a_n \right) \oplus \left(\bigoplus_{\mu_1^{(2)}} a_1 \oplus \cdots \oplus \bigoplus_{\mu_n^{(2)}} a_n \right) \\
&\quad \text{(by \textit{Commutativity} and \textit{Associativity})} \\
&= \biguplus_{i=1}^{n} \left(\bigoplus_{\mu_i^{(1)} - \mu_i^{(2)}} a_i \right) \oplus \biguplus_{i=1}^{n} \left(\bigoplus_{k_i^{(2)}} a_i \right) \\
&\quad \text{(by definition of the summation operator)} \\
&= \biguplus_{i=1}^{n} \left(\bigoplus_{\mu_i^{(1)} - \mu_i^{(2)}} a_i \right) \oplus \biguplus_{i=1}^{n} \left(\bigoplus_{k_i^{(1)}} a_i \right) .
\end{aligned}$$

Hence, the *Cancellation Law* implies that

$$\biguplus_{i=1}^{n} \left(\bigoplus_{\mu_i^{(1)} - \mu_i^{(2)}} a_i \right) = e .$$

Consider any index i, where $1 \leqslant i \leqslant n$. By Property 5.5, $\mu_i^{(1)} - \mu_i^{(2)} = \mu_{i,b}^{(1)} - \mu_{i,b}^{(2)}$. Now, Properties 5.2 and 5.4 immediately imply that $-1 \leqslant \mu_{i,b}^{(1)} - \mu_{i,b}^{(2)} \leqslant 2$, On the other hand, Property 5.6 implies that $\mu_{k,b}^{(1)} - \mu_{k,b}^{(2)} \geqslant 1$, so that not all differences $\mu_{i,b}^{(1)} - \mu_{i,b}^{(2)}$, where

$1 \leqslant i \leqslant n$, are simultaneously zero. It follows that the n elements $a_1, a_2, \ldots, a_n$ are *not* n-wise independent over $\langle \mathbb{\Gamma}, \oplus \rangle$. A contradiction. □

6. The impossibility of finite-size switching networks

Finite-state and infinite-state networks are considered in Sections 6.1 and 6.2, respectively.

6.1. Finite-state networks

We show:

Theorem 6.1 (*Impossibility result for finite-state network*). *There is no non-trivial finite-state switching network $\mathcal{N}$ with concurrency $(or(\mathcal{N}) + 1) \cdot \left(S^{\mathrm{size}(\mathcal{N})} + 1\right)$ that has finite size, incurs register bottleneck at least* 2 *and implements a monotone group* $\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \preccurlyeq \rangle$.

Proof. Assume, by way of contradiction, that there is such a switching network $\mathcal{N}$. Recall that the number of internal configurations of $\mathcal{N}$ is $S^{s(\mathcal{N})}$, where S is the number of internal states of each switch.

Consider a sequential execution α of network $\mathcal{N}$ involving $(or(\mathcal{N}) + 1) \cdot \left(S^{\mathrm{size}(\mathcal{N})} + 1\right)$ tokens, whose types are $(or(\mathcal{N}) + 1) \cdot \left(S^{\mathrm{size}(\mathcal{N})} + 1\right)$-wise independent over $\langle \mathbb{\Gamma}, \oplus, \rangle$. By the *Monotone Linearizability Lemma*, execution α is linearizable. Write $\alpha = \alpha_1 \cdot \alpha_2 \cdot \ldots \cdot \alpha_{\lceil or(\mathcal{N})+1/2 \rceil}$, where each execution fragment α_i, $1 \leqslant i \leqslant \left\lceil or(\mathcal{N}) + \frac{1}{2} \right\rceil$, includes the traversals of $S^{\mathrm{size}(\mathcal{N})} + 1$ tokens.

Take now any execution fragment α_i, where $1 \leqslant i \leqslant (or(\mathcal{N}) + 1)$. Since each token traverses at least one switch, α_i contains at least $S^{\mathrm{size}(\mathcal{N})} + 1$ configurations; so, it contains at least $S^{\mathrm{size}(\mathcal{N})} + 1$ internal configurations. Since the total number of internal configurations of $\mathcal{N}$ is $S^{\mathrm{size}(\mathcal{N})}$, the *Pigeonhole Principle* implies that some internal configuration of $\mathcal{N}$ is repeated in α_i, so that α_i contains at least one pump. Lemma 4.1 implies that there are at least two distinct tokens that access two different output registers in any such pump.

It follows that execution α contains at least $or(\mathcal{N}) + 1$ pumps, and the total number of output registers (allowing repetitions) accessed in these pumps is at least $2 \cdot (or(\mathcal{N}) + 1) > 2or(\mathcal{N})$. The *Pigeonhole Principle* implies that there is at least one output register accessed by tokens in at least three different pumps.

So there are tokens t_1, t_2 and t_3, with $t_1 \xrightarrow{\alpha} t_2 \xrightarrow{\alpha} t_3$, and pumps ϕ_1, ϕ_2 and ϕ_3 of α such that token t_i accesses the same output register r in pump ϕ_i of α, where $1 \leqslant i \leqslant 3$. Consider also output register r' accessed by token t_2' in pump ϕ_2 of α. Since α is a sequential execution, $t_1 \xrightarrow{\alpha} t_2' \xrightarrow{\alpha} t_3$.

We use now execution α to construct another finite (but not sequential) sequence β of alternating configurations and switch transitions, which involves the same tokens as α, with the same types and in the same order, except for the following changes:

> All switch transitions that involve output register r, starting with the one involving token t_1 (and r) and preceding the one involving token t_3 (and r) are scheduled to

occur immediately after the switch transition involving token t_3 (and r), and in the same order (as in α).

So, roughly speaking, all tokens starting with t_1 and not following t_3 that access r are "halted" once they get to cover r and till immediately after token t_3 accesses r.

Clearly, the sequence β is an execution of $\mathcal{N}$, in which each token accesses the same output register as in α.

Since execution α is linearizable and $t_1 \xrightarrow{\alpha} t_2' \xrightarrow{\alpha} t_3$, it follows that $t_1 \xrightarrow{S(\alpha)} t_2' \xrightarrow{S(\alpha)} t_3$. Thus, the *Monotonicity under Composition* property implies that $\mathsf{Out}_\alpha(t_1) \prec \mathsf{Out}_\alpha(t_2') \prec \mathsf{Out}_\alpha(t_3)$.

Since β uses tokens with the same types as α, it follows that the types of the tokens in β are also $\lceil or(\mathcal{N}) + \frac{1}{2} \rceil \cdot \left(S^{\text{size}(\mathcal{N})} + 1\right)$-wise independent over $\langle \mathbb{\Gamma}, \oplus, \rangle$. By the *Monotone Linearizability Lemma*, execution β is linearizable. By construction, $t_2' \xrightarrow{\beta} t_3$. It follows that $t_2' \xrightarrow{S(\beta)} t_3$. Thus, the *Monotonicity under Composition* property implies that $\mathsf{Out}_\beta(t_2') \prec \mathsf{Out}_\beta(t_3)$. However, by construction of β, $\mathsf{Out}_\beta(t_2') = \mathsf{Out}_\alpha(t_2')$, while $\mathsf{Out}_\beta(t_3) = \mathsf{Out}_\alpha(t_1)$. It follows that $\mathsf{Out}_\alpha(t_2') \prec \mathsf{Out}_\alpha(t_1)$. A contradiction. □

We remark that the assumption of non-triviality is essential for Theorem 6.1. Since each token can atomically invoke a computation on an output register, we can implement a monotone RMW operation by a *trivial* switching network consisting of a single switch that outputs tokens along one output wire, which has an associated register that maintains the state of the RMW variable to be implemented. The switch serializes the operations (that correspond to the tokens) so that they can be atomically invoked (by the tokens) on the register.

Recall the *Integers with Addition* monotone group $\langle \mathrm{Z}, \mathbb{N} \setminus \{0\}, +, \leqslant \rangle$ and the *Rationals with Multiplication* monotone group $\langle \mathrm{Z}, \mathbb{N} \setminus \{0\}, +, \leqslant \rangle$, which are associated with the monotone Fetch&Add and Fetch&Multiply operations, respectively. So, Theorem 6.1 immediately implies corresponding impossibility results for switching networks implementing the Fetch&Add and Fetch&Multiply operations.

6.2. *Infinite-state networks*

Clearly, the proof of Theorem 6.1 is not applicable to infinite-state networks since the number of their possible internal configurations is no longer finite. Thus, we need to develop new techniques in order to handle such networks. We show:

Theorem 6.2 (*Impossibility result for infinite-state network*). *There is no non-trivial infinite-state switching network with unbounded concurrency that has finite size, incurs switch bottleneck at least* 2 *and implements a monotone group* $\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \preccurlyeq \rangle$.

Proof. Assume, by way of contradiction that there is such a switching network $\mathcal{N}$. Partition $\mathcal{N}$ into layers $1, 2, \ldots, d(\mathcal{N})$ in the natural way. Assume, without loss of generality, that any switch b at layer ℓ, where $2 \leqslant \ell < d(\mathcal{N})$, has its input wires connected to switches of layer $\ell - 1$ and its output wires connected to switches of layer $\ell + 1$.[3] Since the switch bottleneck of $\mathcal{N}$ is 2, there are at least two switches in each of its layers.

We first construct an infinite non-sequential sequence α for network $\mathcal{N}$. We prepare the reader that α is nearly an execution of network $\mathcal{N}$ since it only fails the liveness condition. However, as we discussed in Section 4.5, α maintains the switch bottleneck of $\mathcal{N}$, and this is all we will need of it. For clarity of exposition, we will abuse terminology and still call α (and several sequences we will derive from it as well) an execution.

Construction of execution α: The execution α involves an infinite sequence of tokens $t_1, t_2, \ldots$ with associated types $a_1, a_2, \ldots$ that are issued by distinct processes. The types are chosen so that for each (finite) prefix $t_1, \ldots, t_n$ of tokens, the associated types $a_1, \ldots, a_n$ are *n*-wise independent over $\langle \mathbb{\Gamma}, \oplus \rangle$.

To construct the execution α, we first define through a simultaneous induction two finite sequences each of length $d(\mathcal{N})$:

- a sequence of pairs of disjoint, infinite subsequences of the type sequence $\mathbf{a} = a_1, a_2, \ldots$, denoted $\langle \mathbf{a}_{1.i}, \mathbf{a}_{2.i} \rangle$, where $1 \leqslant i \leqslant d(\mathcal{N})$;
- a sequence of pairs of distinct switches from the same layer in the network, denoted $\langle b_{1.i}, b_{2.i} \rangle$, where $1 \leqslant i \leqslant d(\mathcal{N})$.

The properties of the two sequences will be used inductively along the way. Specifically, the induction proceeds as follows:

Basis case: Assume that $i = 1$.

- Fix $\mathbf{a}_{1.1}$ and $\mathbf{a}_{2.1}$ to be the odd and even (infinite) subsequences of $\mathbf{a}$, respectively.
- Fix $b_{1.1}$ and $b_{2.1}$ to be any arbitrary switches in layer 1 of the network.

Call tokens in sequences $\mathbf{a}_{1.1}$ and $\mathbf{a}_{2.1}$ the *odd* and *even* tokens, respectively.

Induction hypothesis: Assume that we have defined all pairs $\langle \mathbf{a}_{1.i}, \mathbf{a}_{2.i} \rangle$ and $\langle b_{1.i}, b_{2.i} \rangle$ for all indices *i*, $1 \leqslant i \leqslant k$.

Induction step: We now define $\langle \mathbf{a}_{1.k+1}, \mathbf{a}_{2.k+1} \rangle$ and $\langle b_{1.k+1}, b_{2.k+1} \rangle$.

Since the balancer bottleneck is at least two, the switch $b_{1.k}$ and the infinite sequence $\mathbf{a}_{1.k}$ determine two distinct switches $b_{1.k+1}$ and $b'_{1.k+1}$ and two disjoint infinite sequences $\mathbf{a}_{1.k+1}$ and $\mathbf{a}'_{1.k+1}$ (that are subsequences of $\mathbf{a}_{1.k}$). Correspondingly, the switch $b_{2.k}$ and the infinite sequence $\mathbf{a}_{2.k}$ determine two distinct switches $b_{2.k+1}$ and $b'_{2.k+1}$ and two disjoint infinite sequences $\mathbf{a}_{2.k+1}$ and $\mathbf{a}'_{2.k+1}$ (that are subsequences of $\mathbf{a}_{1.k}$). Assume, without loss of generality, that the switches $b_{1.k+1}$ and $b_{2.k+1}$ are distinct. Note also that the sequences $\mathbf{a}_{1.k+1}$ and $\mathbf{a}_{2.k+1}$ are necessarily disjoint since they are subsequences of $\mathbf{a}_{1.k}$ and $\mathbf{a}_{2.k}$, respectively, which are disjoint by induction hypothesis.

So, for $i = k + 1$, define the pairs $\langle \mathbf{a}_{1.k+1}, \mathbf{a}_{2.k+1} \rangle$ and $\langle b_{1.k+1}, b_{2.k+1} \rangle$, respectively.

Note that our inductive definition guarantees that for each index *i*, where $1 \leqslant i \leqslant d(\mathcal{N})$, the sequences $\mathbf{a}_{1.i}$ and $\mathbf{a}_{2.i}$ contain only odd and even tokens, respectively.

We now continue with the construction of sequence α. Write $\alpha = \alpha_1 \cdot \alpha_2 \cdot \ldots$ as the concatenation of an infinite number of execution fragments, where each execution fragment α_i is finite and includes switch transitions involving token t_i, as follows:

[3] Note that this assumption is indeed with no loss of generality, since for wires that connect non-consecutive layers, we can intercept dummy switches in the missing layers, with input and output width 1, which simply forward tokens (without routing them).

For each token t_i, denote $\mathsf{last}(t_i)$ the largest integer k, $1 \leqslant k < d(\mathcal{N})$, such that token t_i in either $\mathbf{a}_{1.k}$ or $\mathbf{a}_{2.k}$, but in neither $\mathbf{a}_{1.k+1}$ nor $\mathbf{a}_{2.k+1}$, or $d(\mathcal{N})$ if no such integer exists. Then, the execution fragment α_i includes only the switch transitions involving the token t_i and a switch from each layer ℓ, where $1 \leqslant \ell \leqslant \mathsf{last}(t_i)$.

Intuitively, each token t_i enters the network from either switch $b_{1.1}$ or switch $b_{2.1}$; it traverses the network till either it exits the network or it is "halted" once it gets to cover the switch immediately following the switch it has halted in layer $\mathsf{last}(t_i)$ (in case $\mathsf{last}(t_i) < d(\mathcal{N})$).

Note that the construction of execution α guarantees, in particular, that both sequences $\mathbf{a}_{1.d(\mathcal{N})}$ and $\mathbf{a}_{2.d(\mathcal{N})}$ are infinite. Thus, it follows that an infinite number of odd tokens traverses switch $b_{1.d(\mathcal{N})}$, and an infinite number of even tokens traverses switch $b_{2.d(\mathcal{N})}$.

The construction of execution α induces an *odd* path $\pi_1 = b_{1.1}, \ldots, b_{1.d(\mathcal{N})}$ and an *even* path $\pi_2 = b_{2.1}, \ldots, b_{2.d(\mathcal{N})}$. The odd and even paths are traversed by odd and even tokens, respectively. Since the switches $b_{1.i}$ and $b_{2.i}$ are distinct for all layers i, $1 \leqslant i \leqslant d(\mathcal{N})$, it follows that the odd and even paths are disjoint. We prepare the reader that the rest of our proof will use the two possible ways of ordering these two disjoint paths in order to create two corresponding executions. We will use the fact that the two resulting executions are both still indistinguishable from α and linearizable; this will lead to a contradiction. We now continue with the details of the formal proof.

We proceed to use execution α in order to construct a finite execution β.

Construction of execution β: Fix β to be the shortest prefix of α that includes a switch transition involving an odd token at a switch from layer $d(\mathcal{N})$ and a switch transition involving an even token at a switch from layer $d(\mathcal{N})$; thus, β is a (not necessarily complete) finite execution. Since β is finite, it only involves a finite number n of tokens. By construction of execution α, the types of these n tokens are n-wise independent over $\langle \mathbb{\Gamma}, \oplus \rangle$. Moreover, for each token t involved in execution β, $\mathsf{Out}_\beta(t) = \mathsf{Out}_\alpha(t)$.

Denote t_1 and t_2 the latest odd and even tokens, respectively, in execution β. We will use t_1 and t_2 in order to construct from β two distinct finite executions β_1 and β_2.

Construction of executions β_1 *and* β_2: We permute switch transitions in execution β in order to obtain executions β_1 and β_2 as follows:

- In execution β_1, all switch transitions involving odd tokens precede the switch transitions involving even tokens.
- In execution β_1, all switch transitions involving even tokens precede the switch transitions involving odd tokens.

In both executions β_1 and β_2, the relative order of odd tokens (resp., even tokens) is the same as the relative order of odd tokens (resp., even tokens) in execution β.

Since odd tokens (resp., even tokens) follow the odd path π_1 (resp., even path π_2) in both executions β_1 and β_2, the paths π_1 and π_2 are disjoint, and the relative order of odd and even tokens, respectively, is maintained in all executions β, β_1 and β_2, it follows that $\mathsf{Out}_\beta(t_1) = \mathsf{Out}_{\beta_1}(t_1) = \mathsf{Out}_{\beta_2}(t_1)$ and $\mathsf{Out}_\beta(t_2) = \mathsf{Out}_{\beta_1}(t_2) = \mathsf{Out}_{\beta_2}(t_2)$.

We finally use executions β_1 and β_2 in order to construct executions γ_1 and γ_2.

Construction of executions γ_1 *and* γ_2: We extend β_1 and β_2 to complete executions γ_1 and γ_2, respectively.

Since γ_1 extends β_1 and the traversals of tokens t_1 and t_2 are both completed in β_1, it follows that $\mathsf{Out}_{\gamma_1}(t_1) = \mathsf{Out}_{\beta_1}(t_1)$ and $\mathsf{Out}_{\gamma_1}(t_2) = \mathsf{Out}_{\beta_1}(t_2)$. Since γ_2 extends β_2 and the traversals of tokens t_1 and t_2 are both completed in β_2, it follows that $\mathsf{Out}_{\gamma_2}(t_1) = \mathsf{Out}_{\beta_2}(t_1)$ and $\mathsf{Out}_{\gamma_2}(t_2) = \mathsf{Out}_{\beta_2}(t_2)$. It follows that $\mathsf{Out}_{\gamma_1}(t_1) = \mathsf{Out}_{\gamma_2}(t_1) = \mathsf{Out}_{\alpha}(t_1)$ and $\mathsf{Out}_{\gamma_1}(t_2) = \mathsf{Out}_{\gamma_2}(t_2) = \mathsf{Out}_{\alpha}(t_2)$.

Since γ_1 is a complete execution in which the types of operations are n-wise independent, the *Monotone Linearizability Lemma* implies that γ_1 is linearizable. Recall that, by construction, $t_1 \xrightarrow{\gamma_1} t_2$. It follows that $t_1 \xrightarrow{S(\gamma_1)} t_2$. Thus, the *Monotonicity under Composition* property implies that $\mathsf{Out}_{\gamma_1}(t_1) \prec \mathsf{Out}_{\gamma_1}(t_2)$. Similarly, since γ_2 is a complete execution in which the types of operations are n-wise independent, the *Monotone Linearizability Lemma* implies that γ_2 is linearizable. Recall that, by construction, $t_2 \xrightarrow{\gamma_2} t_1$. It follows that $t_2 \xrightarrow{S(\gamma_2)} t_1$. Thus, the *Monotonicity under Composition* property implies that $\mathsf{Out}_{\gamma_2}(t_2) \prec \mathsf{Out}_{\gamma_2}(t_1)$.

So, in total, $\mathsf{Out}_{\gamma_1}(t_1) \prec \mathsf{Out}_{\gamma_1}(t_2) = \mathsf{Out}_{\gamma_2}(t_2) \prec \mathsf{Out}_{\gamma_2}(t_1) = \mathsf{Out}_{\gamma_1}(t_1)$. A contradiction. □

We remark that the assumption of a *non-trivial* switching network is essential for Theorem 6.2 to hold: A switching network consisting of a single infinite-state switch with n input wires and n output wires (where n is the number of concurrent processes) can implement any RMW register as follows. The state of the variable is encoded by the state of the switch. To invoke an operation on the variable, a process issues a token with a state encoding the type of the operation. Such a token, when atomically processed by the switch, will cause the natural changes to its state and to the state of the switch, so that the new state of the switch is the new state of the variable, and the new state of the token is the response of the variable to the operation invoked by the token.

Recall the *Integers with Addition* monotone group $\langle \mathrm{Z}, \mathbb{N} \setminus \{0\}, +, \leqslant \rangle$ and the *Rationals with Multiplication* monotone group $\langle \mathrm{Z}, \mathbb{N} \setminus \{0\}, +, \leqslant \rangle$, which are associated with the monotone Fetch&Add and Fetch&Multiply operations, respectively. So, Theorem 6.1 immediately implies corresponding impossibility results for switching networks implementing the Fetch&Add and Fetch&Multiply operations.

7. Conclusion

We have studied the possibility or impossibility, and the corresponding costs, of devising distributed implementations of any monotone RMW operation that achieve high concurrency and low contention. Through our *Monotone Linearizability Lemma,* which may be of independent interest, we identified inherent ordering constraints of linearizability for any such implementation; we proposed exploiting this inherent linearizability in order to devise impossibility proofs. We succeeded in doing so within the specific context of a switching network implementing a monotone RMW operation, for which we derived the *first* lower bounds on size. These negative end results establish the *first* space complexity separations between Fetch&Increment and any monotone RMW operation in the model of switching networks.

We remark that the proof of the impossibility result for infinite-state networks has required unbounded concurrency. This is not the case for finite-state switching networks, even though we have made similar assumptions on register bottleneck and switch bottleneck for the two classes of switching networks, respectively, in our corresponding proofs. Thus, the two impossibility results represent a trade-off between the strength of the switches (finite or infinite number of states) and the concurrency of the network (bounded or unbounded), and neither of them is implied by the other.

Finally, we mention that we are able to use our *Monotone Linearizability Lemma* to prove a lower bound on latency for switching networks that implement monotone groups. Specifically, we prove that any switching network (whether made up of switches with a finite or infinite number of states) that implements a monotone RMW operation induces executions with latency $\Omega(n)$, where n is the number of concurrent processes participating in the execution. This lower bound complements the corresponding lower bound on latency shown in [7, Theorem 3.2].

Acknowledgements

We would like to thank the anonymous *Theoretical Computer Science* and *SIROCCO 2003* reviewers for their helpful comments.

References

[1] W. Aiello, C. Busch, M. Herlihy, M. Mavronicolas, N. Shavit, D. Touitou, Supporting increment and decrement operations in balancing networks, Chicago J. Theoretical Comput. Sci. 2000-4, December 14, 2000 (electronic).

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

[3] J.E. Burns, N.A. Lynch, Bounds on shared memory for mutual exclusion, Inform. and Comput. 107 (2) (1993) 171–184.

[4] C. Dwork, M. Herlihy, O. Waarts, Contention in shared memory algorithms, J. ACM 44 (6) (1997) 779–805.

[5] K.P. Eswaran, J.N. Gray, R.A. Lorie, I.L. Traiger, The notions of consistency and predicate locks in a database system, Commun. ACM 19 (11) (1976) 624–633.

[6] P. Fatourou, M. Herlihy, Adding networks, Proc. 15th Internat. Symp. on DIStributed Computing, J.L. Welch (Ed.), Lecture Notes in Computer Science, Vol. 2180, Springer, Lisbon, Portugal, October 2001, pp. 330–342.

[7] P. Fatourou, M. Herlihy, Read–Modify–Write networks, Distributed Comput. 17 (2004) 33–46.

[8] J. Goodman, M. Vernon, P. Woest, Efficient synchronization primitives for large-scale, cache-coherent multiprocessors, Proc. 3rd Internat. Conf. on Architectural Support for Programming Languages and Operating Systems, April 1989, pp. 64–75.

[9] M. Herlihy, N. Shavit, O. Waarts, Linearizable counting networks, Distributed Comput. 9 (4) (1996) 193–203.

[10] M. Herlihy, J. Wing, Linearizability: a correctness condition for concurrent objects, ACM Trans. Programming Languages and Systems 12 (3) (1990) 463–492.

[11] C.P. Kruskal, L. Rudolph, M. Snir, Efficient synchronization on multiprocessors with shared memory, Proc. 5th Annu. ACM Symp. on Principles of Distributed Computing, August 1986, pp. 218–228.