Alcom-FT Technical Report Series
ALCOMFT-TR-03-134

The Cost of Concurrent, Low-Contention Read&Modify&Write*

Costas Busch† *Marios Mavronicolas*‡ *Paul Spirakis*§

*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, 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.

†Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY 12180. Part of the work of this author was performed while visiting Department of Computer Science, University of Cyprus. Email: `buschc@cs.rpi.edu`

‡Department of Computer Science, University of Cyprus, Nicosia CY-1678, Cyprus. Part of the work of this author was performed while visiting Faculty of Computer Science, Electrical Engineering and Mathematics, University of Paderborn. Email: `mavronic@ucy.ac.cy`

§Department of Computer Engineering and Informatics, University of Patras, Rion, 265 00 Patras, Greece, & Computer Technology Institute, P. O. Box 1122, 261 10 Patras, Greece. Email: `spirakis@cti.gr`

Abstract

We embark on a study of 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. We consider a natural class of *monotone* RMW operations associated with *monotone groups,* a certain class of algebraic groups introduced here. The popular Fetch&Add and Fetch&Multiply operations are examples from the class.

Our chief combinatorial instrument is a *Monotone Linearizability Lemma,* which establishes inherent ordering constraints of *linearizability* for a certain class of executions of *any* distributed system that implements a monotone RMW operation.

The end results of our study 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* [3] (which implemented the traditional Fetch&Increment operation). These results are negative; they are shown through the *Monotone Linearizability Lemma.* In particular, we derive the *first* lower bounds on *size* (the number of *switches* in the network) and *latency* (the maximum number of switches traversed) for any (non-trivial) switching network implementing a monotone RMW operation:

- If the network incurs low contention, then its size must be infinite. For switches with a finite number of states, *bounded* concurrency (number of concurrent processes) already suffices to prove the lower bound, while a separate proof employs *unbounded* concurrency in order to compensate for switches with an infinite number of states.

  Since Fetch&Increment is implementable with counting networks of *finite* size [3], these lower bounds imply a space complexity separation between Fetch&Increment and any monotone RMW operation in the model of switching networks.

- Any such network has *sequential* executions with n processes and latency $\Omega(n)$.

  Since Fetch&Increment is implementable with counting networks of latency $\Theta(\lg n)$ [11, 23], this lower bound implies a time complexity separation between Fetch&Increment and any monotone RMW operation in the model of switching networks.

Together our lower bounds provide a mathematical explanation for the observed inability of researchers over the last twelve years to extend counting networks, while keeping them finite-sized, high-concurrency, low-latency 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.

1 Introduction

1.1 Background and Motivation

1.1.1 RMW Registers

A Read&Modify&Write *shared variable* or *register* [18, 24], 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 value, and writing the new value back to the register, all in a single, *atomic* (indivisible) RMW operation (cf. [26, Example 9.4.2]). For example, a Fetch&Increment register (over the integers) provides an operation that atomically adds one to its value and returns its prior value; a Fetch&Add register (over the integers) 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. For example, the folklore algorithm for mutual exclusion described in [5, Section 4.3.2] relies on the use of a generic RMW operation that manipulates a FIFO queue; moreover, the recent. scalable *ordered multicast* protocol due to Herlihy *et al.* [21] is based on a modular use of the Swap operation, a natural variant of the RMW operation where the new value to be written is provided as an argument to the operation, and so it is independent of the old value.

Due to their fundamental importance as synchronization primitives, it is most 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 quite convincingly that contention is a critical factor for the overall efficiency of (asynchronous) shared memory algorithms (see, e.g., [13] and references therein). For example, a *counting network* [3] is a particular class of *finite-sized* distributed data structures used to construct high-concurrency, low-latency and low-contention implementations of RMW registers that simultaneously support the Fetch&Increment and Fetch&Decrement operations [2].

1.1.2 Motivation

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 which costs, a generalization of counting networks to implement the *general* RMW operation while still retaining the nice properties of finite size, low latency and low contention?

1.2 Framework and Summary

1.2.1 Monotone RMW Operations

We focus on a specific class of RMW operations whose associated 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 initial one with respect to the total order.

For example, the Fetch&Add operation (over the set of integers) clearly falls into the context of monotone groups, since adding a positive integer to a positive integer results in a larger positive integer; here, the monotone subdomain is the set of positive integers. So also does the Fetch&Multiply operation, and so on. A *monotone* RMW operation is one that is associated with a monotone group.

1.2.2 n-Wise Independence Implies Inherent Linearizability

An abstract concept defined in relation to monotone groups is that of *n-wise independence.* Roughly speaking, n (other than the identity) elements of a monotone group are *n-wise independent* if it is not possible to derive the identity element of the group through some sequence of successive applications of the operator on n of the elements or their inverses.

A significant property of monotone groups we prove is that *every* monotone group is *n-wise independent,* in the sense of having n-wise independent elements. For example, for the monotone group associated with the Fetch&Add monotone operation, we observe that some suitable set of n powers of two comprise a set of n-wise independent elements.

Most importantly, we establish that the existence of n-wise independent elements in a monotone group is largely responsible for enforcing *linearizability* [22] for certain suitable executions of a distributed system that implements the corresponding (monotone) RMW operation; recall that an execution is *linearizable* [22] if the values returned to operations in it respect the real-time ordering of the operations.

1.2.3 Inherent Linearizability Implies Inefficiency

The main conclusion of our work is that the requirement to guarantee the *inherent* linearizability for certain particular executions incurs a high cost in efficiency for a certain class of highly concurrent, low-contention implementations of (monotone) RMW that are based on switching networks. The cost is measured in both *size* and *latency.* See below for more on size, latency, concurrency and the low-contention property for switching networks.

1.2.4 Switching Networks

Switching networks are a class of distributed data strucrures, recently described by Fatourou and Herlihy [15], that may be used for concurrent, low-contention implementations of RMW registers; switching networks are a natural generalization of *counting networks* [3].

Roughly speaking, a *switching network* is a directed, acyclic graph made up of nodes called *switches* and *output registers,* and edges called *wires*. Whenever a process issues a RMW operation, it shepherds a *token* through the network, which traverses a path of switches. Both switches and tokens have internal states. A token arrives at a switch via an *input wire*. In a single atomic step, the switch and the token change their states, and the token leaves the switch on an *output wire*. The token is eventually returned a value when it arrives at an output register.

Clearly, concurrent processes are spatially dispersed in a switching network, which reduces their simultaneous crossings in front of the same memory location (switch or output register). This offers potential for low contention.

1.2.5 Size, Latency, Concurrency, Output Bottleneck and Layer Bottleneck

The *size* of a switching network is the total number of switches in it; its *latency* is the maximum number of switches traversed by a token shepherding a RMW operation through the network. Thus, size and latency are two natural measures for space complexity and time complexity, respectively, in the model of switching networks.

The *concurrency* of a switching network is the maximum number of concurrent *processes* that may simultaneously shepherd a RMW operation through the network.

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

1.3 Contribution and Significance

1.3.1 The Monotone Linearizability Lemma

Our chief combinatorial instrument is a *Monotone Linearizability Lemma* (Proposition 5.2). This establishes inherent ordering constraints of linearizability for a certain class of executions of *any* distributed system that implements a monotone RMW operation. Interestingly, in these executions, the arguments of the RMW operations performed by the n participating concurrent processes enjoy together the group-theoretic property of n-wise independence over the associated monotone group.

In order to gain some intuition for the *Monotone Linearizability Lemma* and its proof, we first offer a substantially simpler proof for a corresponding *Monotone Sequential Consistency Lemma* (Proposition 5.1). In a corresponding way, the *Monotone Sequential Consistency Lemma* establishes inherent ordering constrains of *sequential consistency* [25] for a certain class of executions of *any* distributed system that implements a monotone RMW operation. (Recall that an execution is *sequentially consistent* [25] if the values returned to operations at the *same* process respect the real-time ordering of the operations.)

1.3.2 Proof Methodology for Impossibility Results

The end results of our study are *negative*; they are shown through a modular use of the *Monotone Linearizability Lemma.* The corresponding general methodology we propose for showing impossibility results for any given class of distributed implementations of monotone RMW operations is to devise a suitable counter-example execution so that:

- on one hand, the execution is inherently linearizable, courtesy of the *Monotone Linearizability Lemma,* since the n arguments of the operations issued by the n concurrent processes participating in the execution are n-wise independent (over the associated monotone group);
- on the other hand, the execution is correct (with respect to the values returned to the processes) only if it is *not* linearizable.

This derived contradiction yields the result.

Applying this general methodology to implementations based on switching networks yields the *first* lower bounds on size and latency for any highly concurrent, low-contention switching network that implements a monotone RMW operation.

1.3.3 Impossibility Results for Switching Networks

We obtain the following results for any switching network other than the trivial single-switch one:

- If the switching network is made up of switches with a *finite* number of states and it is low-contention, then it must contain an *infinite* number of switches, even if concurrency is restricted to remain *bounded* (Theorem 6.1).

- If the switching network is made up of switches with an *infinite* number of states and it is low-contention, then it must still contain an *infinite* number of switches if concurrency is now allowed to grow unbounded (Theorem 6.4).

We note that our two lower bounds on the size of any switching network that implements a monotone RMW operation represent a trade-off between the strength of the switches (finite or infinite state) and the concurrency of the network (bounded or unbounded). Thus, neither of them is implied by the other.

Our final impossibility result for switching networks deals with latency. We obtain:

- Any switching network (whether made up of switches of finite or infinite state) that implements a monotone RMW operation induces executions with latency at least $\left\lceil \frac{n-1}{c-1} \right\rceil$, where n is the number of concurrent processes participating in the execution, and c, the network's *capacity,* is the maximum number of processes that simultaneously access a switch in any execution of the network.

1.3.4 The Impact of Inherent Linearizabilty

Our impossibility results for switching networks indicate that inherent linearizability, established through the *Monotone Linearizability Lemma,* is the crucial bottleneck that rules out efficiency (with respect to both size and latency) for any concurrent, low-contention switching network that implements a monotone RMW operation.

In fact, we believe that inherent linearizability is indeed the crucial efficiency bottleneck for *any* such class of concurrent, low-contention implementations, and not just for the class of implementations that are based on switching networks. Of course, this remains to be seen.

Finally, we remark that linearizability has so far been studied as a *required* property for a distributed system that best guarantees acceptable concurrent behavior; see, e.g., [4, 27, 29] and references therein. To the best of our knowledge, our work is the *first* to provide, through the *Monotone Linearizability Lemma,* a (non-trivial) instance of a distributed system where linearizability is an *inherent* property.

1.3.5 Significance of Impossibility Results

Theorems 6.1 and 6.4 settle to the negative a far generalization of a specific open question articulated by Fatourou and Herlihy [15, Section 5] about the existence of switching networks with a *finite* number of switches that implement the (monotone) Fetch&Add operation. Indeed, the more general problem of devising distibuted data structures of *finite* size, as suitable extensions to counting networks, to support synchronization operations other than Fetch&Increment (originally supported by counting networks) was already stated in the seminal work of Aspnes *et al.* [3] that introduced counting networks. However, this problem has remained tantalizingly open; progress on this problem has been so far limited to discovering that counting networks themselves can also support Fetch&Decrement (concurrently with Fetch&Increment)[2, 32].

Counting networks provide concurrent implementations of the Fetch&Increment operation achieving *finite* size and latency *polylogarithmic* in the number n of concurrent processes [3].* In contrast:

- Theorems 6.1 and 6.4 establish that any low-contention switching network for a monotone RMW operation must have infinite size. So, they imply a space complexity separation between Fetch&Increment and any monotone RMW operation in the model of switching networks.

- Theorem 7.1 establishes that any switching network for a monotone RMW operation must incur latency *linear* in the number of concurrent processes; moreover, this happens even in *sequential* executions (in the abscence of concurrency). So, it implies a time complexity separation between Fetch&Increment and any monotone RMW operation in the model of switching networks.

In summary, our impossibility results imply that we *cannot* conveniently generalize counting networks, while still retaining them finite-sized, highly concurrent, low-latency 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 impossibility results are significant since they provide a mathematical explanation for the observed inability of researchers in the last twelve years or so (since the original conference publication of counting networks [3]) to achieve such generalizations.

1.4 Related Work and Comparison

1.4.1 General Related Work

The notion of linearizability has been introduced by Herlihy and Wing [22], extending *atomicity* [30] to arbitrary data types; sequential consistency has been introduced by Lamport [25]. Sequential consistency and linearizability have been studied extensively as required consistency conditions for a distributed system; see, e.g., [4, 27, 29] and references therein.

Counting networks were introduced in the seminal work of Aspnes *et al.* [3]; it was shown in [2, 32] that the Fetch&Decrement operation can be supported by counting networks concurrently with the Fetch&Increment operation originally supported. For more on counting networks,

*The original constructions of counting networks in [3], based on Batcher's *bitonic* sorting network [6] and on the *periodic* sorting network [12], achieve size $\Theta(n \lg^2 n)$ and latency $\Theta(\lg^2 n)$. Improved albeit more involved constructions in [11, 23] achieve size $\Theta(n \lg n)$ and latency $\Theta(\lg n)$.

see, e.g., [9, 10, 20, 27, 29]. Linearizable counting networks were introduced and studied by Herlihy *et al.* [20]; roughly speaking, a counting network is *linearizable* if the values returned to tokens respect their real-time orderings.

Switching networks were recently studied in [15]. In particular, Fatourou and Herlihy [15, Section 4] presented two particular switching networks, called *adding networks,* that implement the monotone Fetch&Add operation; unavoidably, these adding networks contain an *infinite* number of switches. The adding networks presented in [15, Section 4] have the same topology as some corresponding (infinite size) linearizable counting networks presented in [20].

1.4.2 Relation to Work on Linearizable Counting Networks

The structure of the proofs of Theorems 6.1 and 6.4 is inspired by that of the proof of a result of Herlihy *et al.* [20, Theorem 5.1], showing that any (non-blocking) linearizable counting network [3] (other than the trivial single-balancer one) must have an *infinite* number of balancers. The requirement that *all* executions be linearizable allows the proof of [20, Theorem 5.1] to pick any arbitrary execution of choice and force it to violate linearizability. However, a switching network for a monotone RMW operation need not guarantee linearizability in all executions; thus, the role of the *Monotone Linearizability Lemma* is to contribute to the proofs of Theorems 6.1 and 6.4 executions that are necessarily linearizable.

Note also that although a counting network is a special case of a switching network, the lower bound on size established in [20, Theorem 5.1] for a linearizable counting network does not immediately apply to switching networks that implement a monotone RMW operation, since the proof of [20, Theorem 5.1] relies on the behavior of counting networks. Instead, the proofs of Theorems 6.1 and 6.4 require far more delicate arguments that are specific to the behavior of switching networks.

1.4.3 Relation to Work on Switching Networks

Theorem 7.1 is reminiscent of a recent result [15, Theorem 1] that establishes a lower bound of $\left\lceil \frac{n-1}{c-1} \right\rceil$ on latency for adding networks, where there are only two possible arguments for addition, namely a and b such that $|a| > |b| > 0$; more specifically, it is shown that each token of weight b traverses at least $\left\lceil \frac{n-1}{c-1} \right\rceil$ switches, while, if also $|b| > 1$, each token of weight a traverses at least $\left\lceil \frac{n-1}{c-1} \right\rceil$ switches. Theorem 7.1 significantly extends and improves [15, Theorem 1] in the following ways: First, Theorem 7.1 applies to switching networks that implement *any* monotone RMW operation, while [15, Theorem 1] is specific to adding networks and the Fetch&Add operation, and second, despite the enhanced generality of Theorem 7.1, its proof is far simpler and more natural and succinct than that of [15, Theorem 1].

1.5 Road Map

The rest of this paper is organized as follows. Section 2 introduces monotone groups. Definitions for the model of a distributed system we shall consider appear in Section 3. Section 4 provides a framework for switching networks. The *Monotone Linearizability Lemma* is the subject of Section 5. Switching networks implementing monotone groups are studied in Sections 6 and 7. In particular, lower bounds on the size and latency of switching networks implementing a monotone group are shown in Sections 6 and 7, respectively. We conclude, in Section 8, with a discussion of our results and some open problems.

2 Monotone Groups

In this section, we introduce and study monotone groups. This section is organized as follows. Section 2.1 reviews some very basic definitions from Group Theory.† Some composite operators are introduced in Section 2.2. Section 2.3 provides the basic definitions for monotone groups. Pairwise independence is introduced in Section 2.4, while Section 2.5 establishes that all monotone groups are pairwise independent. Similarly, n-wise independence is introduced in Section 2.6, while Section 2.7 establishes that all monotone groups are n-wise independent.

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 $\leq$ the *less-than-or-equal* relation (total order) on these sets.

The reader may prefer to skip this section for now, returning to it later when its results are required.

2.1 Groups and Abelian Groups

A (binary) *operator* (also called *composition law*) on a set Γ is a mapping $\oplus : \Gamma \times \Gamma \rightarrow \Gamma$. A *group* $\langle \Gamma, \oplus \rangle$ is a set Γ together with an operator $\oplus$ such that the following properties hold:

1. *Closure:* For all pairs of elements $a, b \in \Gamma$, $a \oplus b \in \Gamma$.

2. *Associativity:* For all triples of elements $a, b, c \in \Gamma$, $(a \oplus b) \oplus c = a \oplus (b \oplus c)$.

3. *Identity Element:* There is an element $a \in \Gamma$, called the *identity element* of Γ, such that for each element $a \in \Gamma$, $a \oplus e = e \oplus a = a$.

†The interested reader may consult [19] for a general background in Group Theory.

4. *Inverse Element:* for each element $a \in \Gamma$, there is an element $a^{-1} \in \Gamma$, called the *inverse* of a, such that $a \oplus a^{-1} = a^{-1} \oplus a = e$.

An *Abelian group* is a group $\langle \Gamma, \oplus \rangle$ which satisfies the following additional property:

5. *Commutativity:* For all pairs of elements $a, b \in \Gamma$, $a \oplus b = b \oplus a$.

Note that $e^{-1} = e$. Note also that for any sequence of elements $a, b, \ldots \in \Gamma$, the *Associativity* property implies that $(a \oplus b \oplus \ldots \oplus w)^{-1} = w^{-1} \oplus \ldots \oplus b^{-1} \oplus a^{-1}$. Finally, the following elementary property of groups will be used in some of our later proofs.

Property 2.1 (Cancellation Law) *Consider any group $\langle \Gamma, \oplus \rangle$. Then, for any triple of elements $a, b, c \in \Gamma$, $a \oplus b = a \oplus c$ (resp., $b \oplus a = c \oplus a$) implies $b = c$.*

2.2 Composite Operators

We proceed to define two composite operators by applying the (binary) operator $\oplus$ a number of times.

2.2.1 The Power Operator

For any integer k, define the unary operator $\bigoplus_k : \Gamma \to \Gamma$ as follows:

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

Call $\bigoplus_k$ the *power operator*. For any element $a \in \Gamma$, use the power operator $\bigoplus_k$ defined for all integers k to define the set $\Gamma_a = \{\bigoplus_k a \mid k \in \mathrm{Z}\}$. Thus, $\langle \Gamma_a, \oplus \rangle$ is a *cyclic* group with *generator* a.

By the definition for the power operator, it follows that for any element $a \in \Gamma$ and integer k, $\bigoplus_k a = \bigoplus_{-k} a^{-1}$. We continue to prove two elementary properties of the power operator that will be used in some of our later proofs.

Property 2.2 (Superposition of Powers) *For any Abelian group $\langle \Gamma, \oplus \rangle$, fix any element $a \in \Gamma$. Then, for any sequence of integers $k_1, k_2, \ldots, k_n$,*

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

Proof: By definition of the power operator, each factor $\bigoplus_{k_i} a$, $1 \leq i \leq n$, contributes:

- either the element a k_i times if $k_i > 0$ (call these *positive* contributions),
- or the element a^{-1} $-k_i$ times if $k_i < 0$ (call these *negative* contributions),
- or the identity element e if $k_i = 0$ (call these *zero* contributions),

to the composite expression $(\bigoplus_{k_1} a) \oplus (\bigoplus_{k_2} a) \oplus \ldots \oplus (\bigoplus_{k_n} a)$. By the *Commutativity* property, positive, negative and zero contributions can be separated from each other in the composite expression. By definition of the power operator, all zero contributions result to e; by the *Associativity* property, each pair of a positive and a negative contribution cancels out. It follows that the composite expression simplifies to $\bigoplus_{\sum_{i=1}^{n} k_i} a$, as needed. ∎

We finally prove:

Property 2.3 (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, $\bigoplus_k (\bigoplus_l a) = \bigoplus_{k \cdot l} a$.*

Proof: Consider first the case $l = 0$.

- On one hand, $\bigoplus_l a = \bigoplus_0 a = e$ (by definition of the power operator); so,

$$\begin{aligned}
\bigoplus_k \left(\bigoplus_l a \right) &= \bigoplus_k e \\
&= \begin{cases} \underbrace{e \oplus e \oplus \ldots \oplus e}_{k \text{ times}}, & \text{if } k > 0 \\ e\,, & \text{if } k = 0 \\ \underbrace{e^{-1} \oplus e^{-1} \oplus \ldots \oplus e^{-1}}_{-k \text{ times}}, & \text{if } k < 0 \end{cases} \\
&= e\,;
\end{aligned}$$

- On the other hand, $\bigoplus_{k \cdot l} a = \bigoplus_0 a = e$ (by definition of the power operator).

It follows that $\bigoplus_k (\bigoplus_l a) = \bigoplus_{k \cdot l} a$ for $l = 0$. So assume $l > 0$. Clearly,

$$\begin{aligned}
&\bigoplus_k \left(\bigoplus_l a \right) \\
= &\begin{cases} \underbrace{\left(\bigoplus_l a \right) \oplus \left(\bigoplus_l a \right) \oplus \ldots \oplus \left(\bigoplus_l a \right)}_{k \text{ times}}, & \text{if } k > 0 \\ e, & \text{if } k = 0 \\ \underbrace{\left(\bigoplus_l a \right)^{-1} \oplus \left(\bigoplus_l a \right)^{-1} \oplus \ldots \oplus \left(\bigoplus_l a \right)^{-1}}_{-k \text{ times}}, & \text{if } k < 0 \end{cases}
\end{aligned}$$

$$= \begin{cases} \underbrace{\underbrace{(a \oplus a \oplus \ldots \oplus a)}_{l \text{ times}} \oplus \underbrace{(a \oplus a \oplus \ldots \oplus a)}_{l \text{ times}} \oplus \ldots \oplus \underbrace{(a \oplus a \oplus \ldots \oplus a)}_{l \text{ times}}}_{k \text{ times}}, & \text{if } k > 0 \\ e, & \text{if } k = 0 \\ \underbrace{\underbrace{(a \oplus a \oplus \ldots \oplus a)^{-1}}_{l \text{ times}} \oplus \underbrace{(a \oplus a \oplus \ldots \oplus a)^{-1}}_{l \text{ times}} \oplus \ldots \oplus \underbrace{(a \oplus a \oplus \ldots \oplus a)^{-1}}_{l \text{ times}}}_{-k \text{ times}}, & \text{if } k < 0 \end{cases}$$

$$= \begin{cases} \underbrace{a \oplus a \oplus \ldots \oplus a}_{k \cdot l \text{ times}}, & \text{if } k > 0 \\ e, & \text{if } k = 0 \\ \underbrace{\underbrace{(a^{-1} \oplus a^{-1} \oplus \ldots \oplus a^{-1})}_{l \text{ times}} \oplus \underbrace{(a^{-1} \oplus a^{-1} \oplus \ldots \oplus a^{-1})}_{l \text{ times}} \oplus \ldots \oplus \underbrace{(a^{-1} \oplus a^{-1} \oplus \ldots \oplus a^{-1})}_{l \text{ times}}}_{-k \text{ times}}, & \text{if } k < 0 \end{cases}$$

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

$$= \bigoplus_{k \cdot l} a \,,$$

as needed. ∎

2.2.2 The Summation Operator

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 $|n| \geq 2$. $\biguplus_n$ takes as input an ordered multiset of elements $\{a_1, a_2, \ldots, a_{|n|}\} \in \mathbb{\Gamma}$, and it yields the result

$$\biguplus_n \{a_1, a_2, \ldots, a_n\} \quad = \quad \begin{cases} a_1 \oplus a_2 \oplus \ldots \oplus a_{|n|}\,, & \text{if } n \geq 2 \\ a_1^{-1} \oplus a_2^{-1} \oplus \ldots \oplus a_{|n|}^{-1}\,, & \text{if } n \leq -2 \end{cases}$$

  denoted also as $\biguplus_{i=1}^n a_i$. Note that, by associativity, the result of applying the operator is well defined.

Call $\biguplus$ 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 $n \neq 0$,

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

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.*

2.3 Monotone Groups

Assume now that the set $\mathbb{\Gamma}$ is totally ordered;[‡] thus, a *total order* $\preceq$ 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 $a \preceq b$ 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 used in the definition of the monotone subdomain $\mathbb{M}$ implies that $e \notin \mathbb{M}$, so that $\mathbb{M} \subset \mathbb{\Gamma}$. Notice also that the *Monotonicity under Composition* property used in the definition of the monotone subdomain $\mathbb{M}$ implies that $\mathbb{M}$ is necessarily infinite. We are now ready to define monotone groups.

Definition 2.1 (Monotone Group) *A* monotone group *is a quadruple* $\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \preceq \rangle$*, where* $\langle \mathbb{\Gamma}, \oplus \rangle$ *is an Abelian group,* $\preceq$ *is a total order on* $\mathbb{\Gamma}$*, and* $\mathbb{M} \subseteq \mathbb{\Gamma}$ *is a monotone subdomain of* $\mathbb{\Gamma}$*.*

We proceed with some examples of monotone groups that will be used in our later analysis.

[‡]The idea of augmenting the ground set of group with some order (total or partial) has also been followed in defining and studying partially ordered algebraic structures, research on which has burgeoned in the last 50 years. However, the resulting *partially ordered groups* [17] are completely different than the *monotone groups* introduced and studied in this work.

Example 2.1 *The quadruple* $\langle \mathrm{Z}, \mathbb{N} \setminus \{0\}, +, \leq \rangle$.

Clearly, the quadruple $\langle \mathrm{Z}, \mathbb{N} \setminus \{0\}, +, \leq \rangle$ is a monotone group, called *Integers with Addition.* It is associated with the monotone Fetch&Add operation.

- From the definition of the power operator $\bigoplus_k$, for any integer k, we have that for any integer $a \in \mathrm{Z}$, $\bigoplus_k a = k \cdot a$.
- From the definition of the summation operator $\biguplus_{k_1}^{k_2}$, for any pair of integers k_1 and k_2, we have that for any sequence of $k_2 - k_1 + 1$ integers $a_{k_1}, a_{k_1+1}, \ldots, a_{k_2} \in \mathrm{Z}$,

$$\biguplus_{i=k_1}^{k_2} a_i \quad = \quad \sum_{i=k_1}^{k_2} a_i\,.$$

$\square$

Example 2.2 *The quadruple* $\langle \mathrm{Q}, \mathbb{N} \setminus \{0,1\}, \cdot, \leq \rangle$.

Clearly, the quadruple $\langle \mathrm{Q}, \mathbb{N} \setminus \{0,1\}, \cdot, \leq \rangle$ is also a monotone group, called *Rationals with Multiplication.* It is associated with the monotone Fetch&Multiply operation.

- From the definition of the power operator $\bigoplus_k$, for any integer k, we have that for any rational number $a \in \mathrm{Q}$, $\bigoplus_k a = a^k$.
- From the definition of the summation operator $\biguplus$, for any set of n integers $k_1, k_2, \ldots, k_n$, we have that for any set of n rational numbers $a_{k_1}, a_{k_1+1}, \ldots, a_{k_2} \in \mathrm{Q}$,

$$\biguplus_{i=k_1}^{k_2} a_i \quad = \quad \prod_{i=k_1}^{k_2} a_i\,.$$

$\square$

We finally prove an elementary, non-idempotency property of monotone groups that will be used in some of our later proofs.

Property 2.4 (No Idempotent Power) *For any arbitrary monotone group* $\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \preceq \rangle$, *fix any element* $a \in \mathbb{M}$. *Then, for any integer* k, $\bigoplus_k a = e$ *implies* $k = 0$.

Proof: Assume, by way of contradiction, that $k \neq 0$. Consider first the case where $k > 0$. Then,

$$\begin{array}{lll} & \bigoplus_k a & \\ = & \underbrace{a \oplus a \oplus \ldots \oplus a}_{k \text{ times}} & \text{(by definition of the power operator)} \\ \succ & e & \text{(by } \textit{Monotonicity under Composition}\text{)}\ , \end{array}$$

a contradiction.

Consider now the case where $k < 0$. By definition of the power operator and associativity, it follows that

$$\begin{aligned} \underbrace{a^{-1} \oplus a^{-1} \oplus \ldots \oplus a^{-1}}_{-k \text{ times}} &= e \\ &= \left(\underbrace{a^{-1} \oplus a^{-1} \oplus \ldots \oplus a^{-1}}_{-k \text{ times}} \right) \oplus \left(\underbrace{a \oplus a \oplus \ldots \oplus a}_{-k \text{ times}} \right) . \end{aligned}$$

By Property 2.1, it follows that

$$e = \underbrace{a \oplus a \oplus \ldots \oplus a}_{-k \text{ times}} .$$

Since $-k > 0$, we are reduced to the first case, and the proof is now complete. ■

We remark that the proof of Property 2.4 relied on using the *Monotonicity under Composition* property that holds specifically for monotone groups. So, it is no coincidence that Property 2.4 does *not* necessarily hold for a *general* group.

2.4 Pairwise Independence

Consider any two distinct elements $a_1, a_2 \in \mathbb{\Gamma}$ with $a_1, a_2 \neq e$. Say that a_1 and a_2 are *pairwise independent over* $\langle \mathbb{\Gamma}, \oplus \rangle$ if for any integer k, both $a_1 \neq \bigoplus_k a_2$ and $a_2 \neq \bigoplus_k a_1$. Thus, neither a_1 nor a_2 may result by repetitive application of the operator on the other or on the inverse of the other. We are now ready to define a pairwise independent monotone group.

Definition 2.2 (Pairwise Independent Monotone Group) *Say that the monotone group* $\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \preceq \rangle$ *is* pairwise independent *if there are two distinct elements* $a_1, a_2 \in \mathbb{M}$, *with* $a_1, a_2 \neq e$, *that are pairwise independent over* $\langle \mathbb{\Gamma}, \oplus \rangle$.

We continue with two examples of pairwise independent monotone groups.

Example 2.3 *Pairwise independence of the monotone group* $\langle \mathrm{Z}, \mathbb{N} \setminus \{0\}, +, \leq \rangle$.

From the definition of pairwise independence, two integers $a_1, a_2 \in \mathrm{Z}$, with $a_1, a_2 \neq 0$, are pairwise independent in $\langle \mathrm{Z}+ \rangle$ if for any integer k, both $a_1 \neq k \cdot a_2$ and $a_2 \neq k \cdot a_1$. For any integer $a \geq 2$, consider the consecutive natural numbers a and $a+1$; so, $a, a+1 \in \mathbb{N} \setminus \{0\}$. Clearly, for any integer k, both $a \neq k \cdot (a+1)$ and $a + 1 \neq k \cdot a$. It follows that a and $a+1$ are pairwise independent over $\langle \mathrm{Z}, + \rangle$. Hence, the monotone group $\langle \mathrm{Z}, \mathbb{N} \setminus \{0\}, +, \leq \rangle$ is pairwise independent. □

Example 2.4 *Pairwise independence of the monotone group* $\langle \mathrm{Q}, \mathrm{I\!N} \setminus \{0,1\}, \cdot, \leq \rangle$.

From the definition of pairwise independence, two rational numbers $a_1, a_2 \in \mathrm{Q}$, with $a_1, a_2 \neq 1$, are pairwise independent in $\langle \mathrm{Q}, \cdot \rangle$ if for any integer k, both $a_1 \neq a_2^k$ and $a_2 \neq a_1^k$. Consider any pair of distinct prime numbers p_1 and p_2 greater than 1; so $p_1, p_2 \in \mathrm{I\!N} \setminus \{0,1\}$. Clearly, for any integer k, both $p_1 \neq p_2^k$ and $p_2 \neq p_1^k$. It follows that p_1 and p_2 are pairwise independent over $\langle \mathrm{Q}, \cdot \rangle$. Hence, the monotone group $\langle \mathrm{Q}, \mathrm{I\!N} \setminus \{0,1\}, \cdot, \leq \rangle$ is pairwise independent. □

2.5 Pairwise Independence of All Monotone Groups

We prove that *every* monotone group is pairwise independent.

Lemma 2.5 (Every Monotone Group is Pairwise Independent) *Every monotone group* $\langle \mathrm{I\!\Gamma}, \mathrm{I\!M}, \oplus, \preceq \rangle$ *is pairwise independent.*

Proof: Since the monotone group $\langle \mathrm{Z}, \mathrm{I\!N} \setminus \{0\}, +, \leq \rangle$ is pairwise independent (see Example 2.3), there exist *distinct* natural numbers l_1 and l_2 in $\mathrm{I\!N} \setminus \{0\}$ that are pairwise independent over $\langle \mathrm{Z}, + \rangle$. Fix now any arbitrary element $a \in \mathrm{I\!M}$, and consider the elements $\bigoplus_{l_1} a$ and $\bigoplus_{l_2} a$. Clearly, by monotonicity under composition, these two elements are distinct. We will prove that $\bigoplus_{l_1} a$ and $\bigoplus_{l_2} a$ are pairwise independent over $\langle \mathrm{I\!\Gamma}, \oplus \rangle$.

Assume, by way of contradiction, that the elements $\bigoplus_{l_1} a$ and $\bigoplus_{l_2} a$ are *not* pairwise independent over $\langle \mathrm{I\!\Gamma}, \oplus \rangle$. So, there exists some integer k such that either $\bigoplus_{l_1} a = \bigoplus_k (\bigoplus_{l_2} a)$ or $\bigoplus_{l_2} a = \bigoplus_k (\bigoplus_{l_1} a)$. Assume, without loss of generality, that $\bigoplus_{l_1} a = \bigoplus_k (\bigoplus_{l_2} a)$. By Property 2.3, this implies that $\bigoplus_{l_1} a = \bigoplus_{k \cdot l_2} a$. It follows that $(\bigoplus_{l_1} a) \oplus (\bigoplus_{-k \cdot l_2} a) = (\bigoplus_{k \cdot l_2} a) \oplus (\bigoplus_{-k \cdot l_2} a)$. By Property 2.2, this implies that $\bigoplus_{l_1 - k \cdot l_2} a = \bigoplus_{k \cdot l_2 - k \cdot l_2} a$, or $\bigoplus_{l_1 - k \cdot l_2} a = e$. By Property 2.4, it follows that $l_1 - k \cdot l_2 = 0$ or $l_1 = k \cdot l_2$. Thus, the natural numbers l_1 and l_2 are *not* pairwise independent over $\langle \mathrm{Z}, + \rangle$. A contradiction. ■

2.6 n-Wise Independence

Fix any integer $n \geq 2$. Consider any n distinct elements $a_1, a_2, \ldots, a_n \in \mathrm{I\!\Gamma}$ with $a_1, a_2, \ldots, a_n \neq e$. Say that $a_1, a_2, \ldots, a_n$ are *n-wise independent over* $\langle \mathrm{I\!\Gamma}, \oplus \rangle$ if for any sequence of n integers $k_1, k_2, \ldots, k_n$, where $-1 \leq k_i \leq 2$ for $1 \leq i \leq n$, that are *not all simultaneously* zero, $\biguplus_{i=1}^{n} \bigoplus_{k_i} a_i \neq e$. We are now ready to define an n-wise independent monotone group.

Definition 2.3 (n-Wise Independent Monotone Group) *Say that the monotone group* $\langle \mathrm{I\!\Gamma}, \mathrm{I\!M}, \oplus, \preceq \rangle$ *is* n-wise independent *if there are n distinct elements $a_1, a_2, \ldots, a_n \in \mathrm{I\!M}$, with $a_1, a_2, \ldots, a_n \neq e$, that are n-wise independent over $\langle \mathrm{I\!\Gamma}, \oplus \rangle$.*

Note that n-wise independence is not just a trivial generalization (from 2 to n) of pairwise independence, since it imposes constraints on the integers k_i, $1 \leq i \leq n$ (namely, that $-1 \leq k_i \leq 2$). Notice that, in particular, 2-wise independence and pairwise independence are not identical concepts.

From the definition of n-wise independence, n integers $a_1, a_2, \ldots, a_n \in \mathrm{Z}$, where $n \geq 2$, 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 that for any integer $n \geq 2$, the monotone group $\langle \mathrm{Z}, \mathbb{N} \setminus \{0\}, +, \leq \rangle$ is n-wise independent.

Lemma 2.6 *For any integer $n \geq 2$, the monotone group $\langle \mathrm{Z}, \mathbb{N} \setminus \{0\}, +, \leq \rangle$ is n-wise independent.*

Proof: Fix any integer $\ell \geq 0$. 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 $\langle \mathrm{Z}, \mathbb{N} \setminus \{0\}, +, \leq \rangle$ 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 $k_i \leq 2$ for all indices i, $1 \leq i \leq n-1$,

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

  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 $k_i \geq -1$ for all indices i, $1 \leq i \leq n-1$,

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

  a contradiction.

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

We finally prove:

Lemma 2.7 *For any integer $n \geq 2$, the monotone group $\langle \mathrm{Q}, \mathbb{N} \setminus \{0, 1\}, \cdot, \leq \rangle$ is n-wise independent.*

Proof: From the definition of n-wise independence, n rational numbers $a_1, a_2, \ldots, a_n \in \mathrm{Q}$ are n-wise independent over $\langle \mathrm{Q}, \cdot \rangle$ 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, $\prod_{i=1}^{n} a_i^{k_i} \neq 1$.

Consider any n distinct prime numbers $p_1, p_2, \ldots, p_n \in \mathbb{N} \setminus \{0, 1\}$. Then, $\prod_{i=1}^{n} p_i^{k_i}$ is a rational number whose numerator and denominator have no common factors; so $\prod_{i=1}^{n} p_i^{k_i} \neq 1$. Thus, the n prime numbers $p_1, p_2, \ldots, p_n$ are n-wise independent over Q. Hence, the monotone group $\langle \mathrm{Q}, \mathbb{N} \setminus \{0, 1\}, \cdot, \leq \rangle$ is n-wise independent, as needed. ∎

2.7 n-Wise Independence of All Monotone Groups

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

Lemma 2.8 (Every Monotone Group is n-Wise Independent) *For any integer $n \geq 2$, the monotone group $\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \preceq \rangle$ is n-wise independent.*

Proof: Since the monotone group $\langle \mathrm{Q}, \mathbb{N} \setminus \{0\}, +, \leq \rangle$ is n-wise independent (Lemma 2.6), 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 $\langle \Gamma, \mathbb{M}, \oplus, \preceq \rangle$, these n elements are distinct. We will prove that they are also n-wise independent over $\langle \Gamma, \oplus \rangle$.

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 $\langle \Gamma, \oplus \rangle$. 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

$$\biguplus_{i=1}^{n} \left(\bigoplus_{k_i} \left(\bigoplus_{l_i} a \right) \right) \quad = \quad e\,.$$

By Property 2.3, it follows that

$$\biguplus_{i=1}^{n} \left(\bigoplus_{k_i \cdot l_i} a \right) \quad = \quad e\,,$$

which, by the definition of the summation operator, may be written as

$$\left(\bigoplus_{k_1 \cdot l_1} a \right) \oplus \left(\bigoplus_{k_2 \cdot l_2} a \right) \oplus \ldots \oplus \left(\bigoplus_{k_n \cdot l_n} \right) \quad = \quad e\,.$$

By Property 2.2, it follows that

$$\bigoplus_{\sum_{i=1}^{n} k_i \cdot l_i} a \quad = \quad e\,.$$

Property 2.4, now implies that $\sum_{i=1}^{n} k_i \cdot l_i = 0$. Since the integers k_i, $1 \leq i \leq n$, 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 2.8 employs the n-wise independence of the monotone group $\langle \mathrm{Q}, \mathbb{N} \setminus \{0, \}, +, \leq \rangle$ (which was established in Lemma 2.6) in order to conclude the n-wise independence of the arbitrary monotone group $\langle \Gamma, \mathbb{M}, \oplus, \leq \rangle$. So, this proof by reduction indicates some kind of completeness of the monotone group $\langle \mathrm{Q}, \mathbb{N} \setminus \{0, \}, +, \leq \rangle$ for the class of all n-wise independent monotone groups.

3 System Model

In this section, we provide our definitions for the system model we shall consider. This section is organized as follows. Section 3.1 provides basic definitions for a distributed system that implements a monotone group. Definitions related to sequential consistency and linearizability for such a system 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 [22, Section 2]; however, that one is adjusted in order to incorporate the issue of implementing a monotone group $\langle \mathbb{T}, \mathbb{M}, \oplus, \preceq \rangle$.

We consider a distributed system $\mathbf{P}$ 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 $\mathbf{P}$ 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.

An execution α induces a partial order $\xrightarrow{\alpha}$ on the set of operations in α as follows.

For any pair of 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 $op_{i_1} \stackrel{\alpha}{\not\longrightarrow} op_{i_2}$. If simultaneously $op_{i_1} \stackrel{\alpha}{\not\longrightarrow} op_{i_2}$ and $op_{i_2} \stackrel{\alpha}{\not\longrightarrow} op_{i_1}$, then we say that op_{i_1} and op_{i_2} are *parallel* in execution α, denotedk as $op_{i_1} \parallel_\alpha op_{i_2}$.

For any execution α of system $\mathbf{P}$, a *serialization* $S(\alpha)$ [14] 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 $\stackrel{S(\alpha)}{\longrightarrow}$ 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* $\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \preceq \rangle$ 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. We are now ready to provide an important definition.

Definition 3.1 (Implementation of Monotone Group) System $\mathbf{P}$ implements the monotone group $\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \preceq \rangle$ *if every execution* α *of* $\mathbf{P}$ *has a serialization that is valid for the monotone group.*

Denote $\mathsf{RMW}\,(\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \preceq \rangle)$ the $\mathsf{Read\&Modify\&Write}$ operation associated with the monotone group $\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \preceq \rangle$ in the natural way. So, in particular, $\mathsf{RMW}\,(\langle \mathrm{Z}, \mathbb{N} \setminus \{0\}, +, \leq \rangle)$ and $\mathsf{RMW}\,(\langle \mathrm{Q}, \mathbb{N} \setminus \{0, 1\}, \cdot, \leq \rangle)$ are the RMW operations $\mathsf{Fetch\&Add}$ and $\mathsf{Fetch\&Multiply}$, respectively. *Monotone* RMW operations are those associated with monotone groups. Say that system $\mathbf{P}$ implements the (monotone) operation $\mathsf{RMW}\,(\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \preceq \rangle)$ whenever it implements the associated monotone group.

We continue to state and prove the *Unique Serialization Lemma*; it asserts that for any execution α of a system $\mathbf{P}$ implementing a monotone group, there is a *unique* serialization $S(\alpha)$ that is valid (for the monotone group).

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

Proof: Assume, by way of contradiction, that there are two distinct valid serializations

$$S^{(1)}(\alpha) \quad = \quad op^{(1.1)}, op^{(1.2)}, op^{(1.3)}, \ldots$$

and

$$S^{(2)}(\alpha) \quad = \quad op^{(2.1)}, op^{(2.2)}, op^{(2.3)}, \ldots$$

of execution α. Since $S^{(1)}(\alpha)$ and $S^{(2)}(\alpha)$ are distinct, there exists a *least* index $k \geq 1$ 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

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

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

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

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

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

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

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

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

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

By Property 2.1, it follows that

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

The *Monotonicity under Composition* property implies that

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

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

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

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 using the required properties for a monotone group, namely on the *Monotonicity under Composition* and *Identity Lower Bound* properties. Since these properties do not necessarily hold for a *general* group (one that is not necessarily monotone), the same follows for the *Unique Serialization Lemma.*

In our later definitions and proofs, we will sometimes write $\mathsf{In}_\alpha(op)$ and $\mathsf{Out}_\alpha(op)$ in order to emphasize reference to execution α.

We conclude this section with an immediate consequence of the *Valid Start* and *Valid Composition* conditions assumed in Definition 3.1.

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

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

3.2 Definitions Related to Sequential Consistency and Linearizability

Our definitions adjust previous definitions related to sequential consistency and linearizability [22, 25] to the setting of a distributed system implementing a monotone group.

Throughout this section, we refer to a distributed system $\mathbf{P}$ implementing a monotone group $\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \preceq \rangle$. Our particular definitions will refer to any arbitrary execution α of the system $\mathbf{P}$, and to its valid serialization $S(\alpha)$, which is unique by Lemma 3.1.

3.2.1 Sequential Consistency

Say that *a process p_i is sequentially consistent in execution α* [25] if the serialization $S(\alpha)$ extends $\xrightarrow{\alpha}_i$; that is, for any pair of operations $op_i^{(1)}$ and $op_i^{(2)}$ such that $op_i^{(1)} \xrightarrow{\alpha}_i op_i^{(2)}$, $op_i^{(1)} \xrightarrow{S(\alpha)} op_i^{(2)}$. The *Valid Composition* condition implies that for any two operations $op_i^{(1)}$ and

$op_i^{(2)}$ such that $op_i^{(1)} \xrightarrow{S(\alpha)} op_i^{(2)}$, $\mathsf{Out}(op_i^{(1)}) \prec \mathsf{Out}(op_i^{(2)})$. Thus, it follows that for any process p_i that is sequentially consistent in execution α, for any pair of operations $op_i^{(1)}$ and $op_i^{(2)}$ such that $op_i^{(1)} \xrightarrow{\alpha} op_i^{(2)}$, $\mathsf{Out}(op_i^{(1)}) \prec \mathsf{Out}(op_i^{(2)})$.

Say that operation $op_i^{(1)}$ at process p_i in execution α is *sequentially inconsistent in execution* α if there is another operation $op_i^{(2)}$ at the same process in execution α such that $op_i^{(2)} \xrightarrow{\alpha} op_i^{(1)}$ while $op_i^{(2)} \not\xrightarrow{S(\alpha)} op_i^{(1)}$. Say that operation $op_i^{(1)}$ at process p_i in execution α is *sequentially consistent in execution* α if it is not sequentially inconsistent in execution α. Clearly, process p_i is sequentially consistent in execution α if every operation op_i at process p_i in execution α is sequentially consistent in it.

Say that execution α is *sequentially consistent* [25] if every process p_i is sequentially consistent in α. It follows that execution α is sequentially consistent if every operation in execution α is sequentially consistent in it. Finally, say that the system $\mathbf{P}$ is *sequentially consistent* if all its executions are.

3.2.2 Linearizability

Say that execution α is *linearizable* [22] 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 $op^{(1)} \xrightarrow{S(\alpha)} op^{(2)}$, $\mathsf{Out}(op^{(1)}) \prec \mathsf{Out}(op^{(2)})$. Thus, it follows that for any pair of operations $op^{(1)}$ and $op^{(2)}$ such that $op^{(1)} \xrightarrow{\alpha} op^{(2)}$, $\mathsf{Out}(op^{(1)}) \prec \mathsf{Out}(op^{(2)})$.

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 $op^{(2)} \not\xrightarrow{S(\alpha)} op^{(1)}$. 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, say that system $\mathbf{P}$ 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 [15, Section 2], while most of them refine and extend corresponding ones from [15, Section 2]. This section is organized as follows. Basic definitions about switching networks are articulated in Section 4.1. Processes, tokens, switches and wires are described in Section 4.2. Section 4.3 defines states and configurations for a switching network; executions of a switching network are defined in Section 4.4. Switching networks with finite and infinite switches are introduced and compared in Section 4.5. Section 4.6 introduces some contention measures for switching networks.

4.1 Basic Definitions

A *switching network* [15], like a *counting network* [3], 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_{in}, f_{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* or *infinite* switch, respectively; see Section 4.5 for a finer distinction between these two cases. 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 switches; an *infinite-state switching network* is a switching network made up from infinite switches.

A (w_{in}, w_{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* of a switching network is the total number of its switches. Denote $s(\mathcal{N})$ the size of switching network $\mathcal{N}$. A network $\mathcal{N}$ has *finite* size if $s(\mathcal{N}) < \infty$; else, it has *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 $0 \leq \ell \leq d(\mathcal{N})$. A *path* in a switching network is a sequence of switches each (other than the last) connected to the next.

A simple $(4,4)$-switching network $\mathcal{N}$, made from $(2,2)$-switches and having depth 3, is depicted in Figure 1. Switches are drawn with vertical lines, while wires are drawn with horizontal lines; for each switch, inputs wires appear on its left, while output wires appear on its right, and similarly for the switching network $\mathcal{N}$. Each switch in the network $\mathcal{N}$ is *binary* since it contains only two input wires and two output wires.

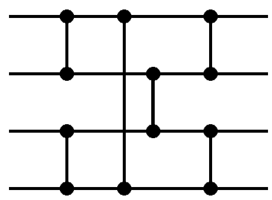

Figure 1: The switching network $\mathcal{N}$

4.2 Processes, Tokens, Switches and Wires

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 [3], 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.
- The switch changes state.
- The token changes state.
- The token is routed to an output wire of the switch.

Note that the token and the switch are allowed to have different transition functions for changing their states.

Example 4.1 *An* (f_{in}, f_{out}) balancer [1].

This is a finite 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_{out} mod k. Thus, the state of an (f_{in}, f_{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_{in}, f_{out})-balancer is not affected. Such balancers have been used to construct counting networks (see, e.g., [1, 3, 9, 11, 20, 23]). $\square$

Each *wire* is a one-way, reliable communication channel that allows reordering of tokens; so, a token does not wait on a wire forever.

4.3 States and Configurations

For each (f_{in}, f_{out})-switch, denote by x_i, $0 \leq i \leq f_{in} - 1$, 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

$$\sum_{i=1}^{f_{in}} x_i = \sum_{j=1}^{f_{out}} y_j \, .$$

A switch satisfies the following two conditions:

1. *Safety condition:* In any state,

$$\sum_{i=1}^{f_{in}} x_i \geq \sum_{j=1}^{f_{out}} y_j \, .$$

   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. Furthermore. the safety and liveness properties for switches and switching networks naturally generalize those for balancers and *balancing networks* [3].

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

4.4 Executions

Herlihy *et al.* [20] modeled executions of counting networks; we follow their style in modeling executions of switching networks.

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 in the style of Lynch and Tuttle [28].

An *execution* of a switching network is a finite or infinite sequence

$$\alpha \quad = \quad Q_0, \tau_1, Q_1, \tau_2, Q_2, \ldots ,$$

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 $i \geq 0$, 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. Loosely speaking, tokens traverse the network one completely after the other in a sequential execution.

An *execution suffix* of a switching network is a finite or infinite sequence

$$\alpha \quad = \quad Q, \tau_1, Q_1, \tau_2, Q_2, \ldots ,$$

of alternating configurations and switch transitions such that:

1. Q is an arbitrary configuration.
2. For each triple $\langle Q_i, \tau_{i+1}, Q_{i+1} \rangle$, where $i \geq 0$, the switch transition τ_{i+1} carries the configuration Q_i to the configuration Q_{i+1}.

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.

Consider now a configuration Q of the switching network $\mathcal{N}$ in which there is a *single* token t on some input wire of the network. Clearly, there is a *unique* execution suffix α of the network $\mathcal{N}$ that starts with the configuration Q. In the execution suffix α, token t follows a path π from the input wire i to an output wire of the network, where it exits the network and it is returned b as its output value. Call π and b the *preferred path* and *preferred value,* respectively, of the token t with respect to the configuration Q.

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; in this case, this number can be either finite or infinite.

The *latency* of a switching network is the maximum number of switches traversed by any token, where the maximum is taken over all executions of the network.

4.5 Finite and Infinite Switches

In Sections 4.5.1 and 4.5.2, we examine finite-state and infinite-state switching networks, respectively.

4.5.1 Switching Networks with Finite Switches

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.

The *output value* for a token is computed on the output register residing on the network's output wire from which the token exits. In more detail, 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 [3], 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.

Denote $or(\mathcal{N})$ the number of output registers in a finite-state switching network $\mathcal{N}$.

4.5.2 Switching Networks with Infinite Switches

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.

For both finite-state and infinite-state switching networks, 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.

4.6 Contention Measures

In a switching network, contention represents the extent to which concurrent processes access the same switch or output register simultaneously. We use three complexity-theoretic measures to model contention in switching networks; the first two of them, namely *register bottleneck* and *layer bottleneck,* are introduced here for the first time, while the last, namely *capacity,* was originally introduced by Dwork *et al.* [13] in the context of counting networks.

4.6.1 Register Bottleneck

This definition applies only to finite-state switching networks.

Definition 4.1 (Register Bottleneck) *The* register bottleneck *of a 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”.

4.6.2 Layer Bottleneck

This definition will be useful for infinite-state switching networks.

Definition 4.2 (Layer Bottleneck) *The* layer bottleneck *of a switching network $\mathcal{N}$ is the* minimum *number of switches in the same layer, where the minimum is taken over all layers and infinite executions of the network, that are accessed by tokens in some infinite suffix of an infinite execution the network.*

On the account of layer bottleneck, a switching network is *low-contention* if its layer bottleneck is sufficiently large. A layer bottleneck of 1 is the *worst* possible layer 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 switch, which thus becomes a “hot-spot”.

4.6.3 Capacity

Definition 4.3 (Capacity) *The* capacity *c of a switching network $\mathcal{N}$ is the* maximum *number of processes, where the maximum is taken over all executions of the network, that simultaneously access a particular switch in an execution of the network.*

On the account of capacity, a switching network is *low-contention* if capacity is sufficiently small. A capacity of n, where n is the number of concurrent processes, is the *worst* possible capacity, since it implies the existence of some execution of the network in which all n concurrent processes will eventually accumulate in front of the same switch, which will thus become a ”hot-spot”.

4.7 Switching Networks Implementing Monotone Groups

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

- Each token t issued by process p_i corresponds to an operation

$$op_i \quad = \quad [\texttt{Invoke}_i(a), \texttt{Response}_i(b)]$$

  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* 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.

Use now execution α to define its subsequence α' that contains only transitions that correspond to invocations and responses of the operations corresponding to tokens. The sequence α' induces an execution of a distributed system in the natural way. Denote $\mathbf{P}(\mathcal{N})$ the induced distributed system that is determined by all such induced executions, one for each execution of the switching network $\mathcal{N}$. We are now ready to provide an important definition.

Definition 4.4 (Switching Network Implementing a Monotone Group) *The* switching network $\mathcal{N}$ implements the monotone group $\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \preceq \rangle$ *if the induced distributed system* $\mathbf{P}(\mathcal{N})$ *implements the monotone group* $\langle \mathbb{\Gamma}, \mathbb{M}, \oplus, \preceq \rangle$.

4.8 Techniques and Tools

In some of our impossibility proofs, we will use a variant of the *variable covering* technique originally introduced by Burns and Lynch [7] for proving lower bounds on the number of read/write registers needed to solve deadlock-free mutual exclusion. Roughly speaking, a token covers a switch if it is about to access the switch. This intuition is captured in the following definition.

Definition 4.5 (Token Covers a Switch) *A token t* covers *a switch s in configuration Q_l of execution α if $\pi_{l+1} = \langle t, s \rangle$.*

Definition 4.5 can be immediately extended to tokens covering output registers as follows.

> Say that a token t *covers* an output register r in configuration Q_l of execution α if it covers the corresponding output switch t of the network.

We finally prove:

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

Proof: This holds since, otherwise, we would be able to construct an infinite execution with register bottleneck 1 by repeating ϕ infinitely many times, so that only one output register would be accessed in the resulting infinite suffix. ■

Lemma 4.2 (Intersection Lemma) *Consider a switching network $\mathcal{N}$ that implements a monotone group. Then, For any arbitrary configuration Q, the preferred paths of any two tokens with respect to configuration Q intersect each other.*

Proof: Consider the network $\mathcal{N}$ in a quiescent configuration s. Denote v the output value returned by $\mathcal{N}$ to the last token in the (unique) valid serialization of the execution fragment ending with total configuration s, and let a denote that last token's input value. Consider tokens t_i and t_j with input values a_i and a_j, respectively. Assume, by way of contradiction, that the preferred paths of t_i and t_j starting from total configuration s do not intersect. Since the network $\mathcal{N}$ implements the monotone group $\langle \mathbb{I}\Gamma, \mathbb{M}, \oplus, \preceq \rangle$, and since the preferred paths do not intersect, the output values of t_i and t_j when they run sequentially into the network $\mathcal{N}$ with t_i first and t_j next, starting from s are equal to $v \oplus a$ (those that would be returned in separate executions where only one of the tokens would be running). However, execution α', where both tokens are running, is linearizable (by Proposition 5.2). Hence, the token t_i is serialized before token t_j in the (unique) valid serialization of execution α'. Since the network $\mathcal{N}$ implements the monotone group $\langle \mathbb{I}\Gamma, \mathbb{M}, \oplus, \preceq \rangle$, the output values of t_i and t_j are $v \oplus a$ and $v \oplus a \oplus a_i$, respectively. Since $a_i \neq e$, $v \oplus a \neq v \oplus a \oplus a_i$, a contradiction. It follows that the preferred paths of t_i and t_j starting from total configuration s do intersect, as claimed. ■

5 The Monotone Linearizability Lemma

Throughout this section, we refer to a distributed system **P** implementing a monotone group $\langle \mathbb{I}\Gamma, \mathbb{M}, \oplus, \preceq \rangle$. 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**. However, in order to provide the reader with useful intuition for the *Monotone Linearizability Lemma* and its proof, we first prove a corresponding *Monotone Sequential Consistency Lemma,* which establishes similar ordering constraints of sequential consistency on the system **P**. The proof of the *Monotone Sequential Consistency Lemma* is substantially simpler and more succinct than the one of the *Monotone Linearizability Lemma* that will follow.

5.1 The Monotone Sequential Consistency Lemma and its Proof

Recall that, by Lemma 2.5, the monotone group $\langle \mathbb{I}\Gamma, \mathbb{M}, \oplus, \preceq \rangle$ is pairwise independent. So, there are two distinct elements $a_1, a_2 \in \mathbb{M}$, with $a_1, a_2 \neq e$, that are pairwise independent over $\langle \mathbb{I}\Gamma, \oplus \rangle$. The proof of the *Monotone Sequential Consistency Lemma* amounts to establishing a contradiction to pairwise independence for a hypothetical *non-sequentially consistent* execution, in which the types of the RMW operations issued by the processes are a_1 and a_2. We are now ready to state and prove the *Monotone Sequential Consistency Lemma.*

Proposition 5.1 (Monotone Sequential Consistency Lemma) *Consider an execution α of system $\mathbf{P}$ in which process p_1 issues only operations of type a_1, while any other process p_i, $i \neq 1$, issues only operations of type a_2. Then, p_1 is sequentially consistent in execution α.*

Proof: We start with an informal outline of our proof. We will proceed by contradiction. So, we will consider the earliest sequentially inconsistent operation $op_1^{(1)}$ at process p_1, and the latest operation $op_1^{(1)}$ (at p_1) that precedes it. We will use these operations to construct two executions γ_1 and γ_2 that are *indistinguishable* to process p_1 with respect to operation $op_1^{(2)}$. This indistinguishability implies that $op_1^{(2)}$ receives the same output in these two executions. Then, 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 pairwise independence. We now continue with the details of the formal proof.

Assume, by way of contradiction, that process p_1 is *not* sequentially consistent in execution α. So, there is at least one operation at p_1 that is sequentially inconsistent in execution α. Consider the *earliest* such operation $op_1^{(1)}$, and let $op_1^{(2)}$ be the *latest* operation at process p_1 that precedes $op_1^{(1)}$ in α. So, $op_1^{(2)} \xrightarrow{\alpha} op_1^{(1)}$ while $op_1^{(1)} \xrightarrow{S(\alpha)} op_1^{(2)}$, where $S(\alpha)$ is the (unique) valid serialization of α.

Denote $k \geq 0$ the number of operations at p_1 that precede $op_1^{(2)}$ in execution α. Since all these operations are sequentially consistent in execution α, they precede $op_1^{(2)}$ in $S(\alpha)$ as well.

In our proof, we will use the operations $op_1^{(1)}$ and $op_1^{(2)}$ 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 $op_1^{(1)}$, and
- the finite prefix β_2 of execution α that ends with the response for operation $op_1^{(2)}$.

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 its corresponding extension; we then treat them together.

<u>Properties of the prefix β_1 and its extension γ_1:</u>

Consider a finite execution γ_1, which is an extension of β_1 that includes no additional invocations by processes; so, γ_1 is an extension of β_1 that additionally includes only 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 $op_1^{(1)}$) have identical outputs in α and γ_1. In particular, $\mathsf{Out}_\alpha\left(op_1^{(1)}\right) = \mathsf{Out}_{\gamma_1}\left(op_1^{(1)}\right)$ and $\mathsf{Out}_\alpha\left(op_1^{(2)}\right) = \mathsf{Out}_{\gamma_1}\left(op_1^{(2)}\right)$. Take now the (unique) valid serialization $S(\gamma_1)$ of γ_1.

Since $op_1^{(1)} \xrightarrow{S(\alpha)} op_1^{(2)}$, the *Valid Composition* condition (for $S(\alpha)$) implies that $\mathsf{Out}_\alpha\left(op_1^{(1)}\right) \prec \mathsf{Out}_\alpha\left(op_1^{(2)}\right)$. Since $\mathsf{Out}_\alpha\left(op_1^{(1)}\right) = \mathsf{Out}_{\gamma_1}\left(op_1^{(1)}\right)$ and $\mathsf{Out}_\alpha\left(op_1^{(2)}\right) = \mathsf{Out}_{\gamma_1}\left(op_1^{(2)}\right)$, it follows that $\mathsf{Out}_{\gamma_1}\left(op_1^{(1)}\right) \prec \mathsf{Out}_{\gamma_1}\left(op_1^{(2)}\right)$. The *Valid Composition* condition (for $S(\gamma_1)$) implies now that $op_1^{(1)} \xrightarrow{S(\gamma_1)} op_1^{(2)}$,

Since the outputs of the k operations (of type a_1) at p_1 that precede $op_1^{(2)}$ in execution α. are identical in α and γ_1, the *Valid Composition* condition (for γ_1) implies that all these operations precede $op_1^{(2)}$ in $S(\gamma_1)$ as well.

Denote l_1 the number of operations at other processes that precede $op_1^{(2)}$ in the serialization $S(\gamma_1)$; recall that all these operations are of type a_2.

By Property 3.2, $\mathsf{Out}_{\gamma_1}\left(op_1^{(2)}\right)$ is a composite expression involving $k+1$ contributions of a_1 and l_1 contributions of a_2. By the *Commutativity* property, these two types of contributions can be separated from each other in the composite expression, so that

$$\mathsf{Out}_{\gamma_1}\left(op_1^{(2)}\right) \quad = \quad \bigoplus_{k+1} a_1 \oplus \bigoplus_{l_1} a_2\,.$$

<u>Properties of the prefix β_2 and its extension γ_2:</u>

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.

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 $op_1^{(2)}$) have identical outputs in α and γ_2. In particular, $\mathsf{Out}_\alpha\left(op_1^{(2)}\right) = \mathsf{Out}_{\gamma_2}\left(op_1^{(2)}\right)$. Take now the (unique) valid serialization $S(\gamma_2)$ of execution γ_2.

Since the outputs of the k operations (of type a_1) at p_1 that precede $op_1^{(2)}$ in execution α. are identical in α and γ_2, the *Valid Composition* condition (for γ_2) implies that all these operations precede $op_1^{(2)}$ in $S(\gamma_2)$ as well.

Denote l_2 the number of operations (all of type a_2) at other processes that precede $op_1^{(2)}$ in the serialization $S(\gamma_2)$.

By Property 3.2, $\mathsf{Out}_{\gamma_2}\left(op_1^{(2)}\right)$ is a composite expression involving k contributions of a_1 and l_2 contributions of a_2. By the *Commutativity* property, these two types of contributions can be separated from each other in the composite expression, so that

$$\mathsf{Out}_{\gamma_2}\left(op_1^{(2)}\right) \quad = \quad \bigoplus_k a_1 \oplus \bigoplus_{l_2} a_2\,.$$

Joint properties of the prefixes β_1 and β_2 and their extensions γ_1 and γ_2: Recall that $\mathsf{Out}_{\alpha}\left(op_1^{(2)}\right) = \mathsf{Out}_{\gamma_1}\left(op_1^{(2)}\right)$ and $\mathsf{Out}_{\alpha}\left(op_1^{(2)}\right) = \mathsf{Out}_{\gamma_2}\left(op_1^{(2)}\right)$; hence, it follows that $\mathsf{Out}_{\gamma_1}\left(op_1^{(2)}\right) = \mathsf{Out}_{\gamma_2}\left(op_1^{(2)}\right)$, so that

$$\bigoplus_{k+1} a_1 \oplus \bigoplus_{l_1} a_2 \quad = \quad \bigoplus_k a_1 \oplus \bigoplus_{l_2} a_2\,.$$

By Property 2.2, $\bigoplus_{l_2} a_2 = \bigoplus_{l_2-l_1} a_2 \oplus \bigoplus_{l_1} a_2$. Hence, by Property 2.1, it follows that $a_1 = \bigoplus_{l_2-l_1} a_2$. So, a_1 and a_2 are *not* pairwise independent over $\langle \mathbb{I}\Gamma, \oplus \rangle$. A contradiction. ■

5.2 The Monotone Linearizability Lemma and its Proof

Recall that, by Lemma 2.8, the monotone group $\langle \mathbb{I}\Gamma, \mathbb{M}, \oplus, \preceq \rangle$ is n-wise independent for any integer $n \geq 2$. 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 $\langle \mathbb{I}\Gamma, \oplus \rangle$. 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 5.2 (Monotone Linearizability Lemma) *Consider any execution α of system $\mathbf{P}$ in which each process p_i issues only operations of type a_i, where $1 \leq i \leq n$. 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.

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 $1 \leq i \leq n$, 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 5.3 *For each process p_i, where $1 \leq i \leq n$, $\mu_{i,b}^{(1)} \leq 2$.*

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 $\mu_{i,b}^{(1)} \leq 1$ 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 $\mathsf{Out}_{\gamma_1}(op') = \mathsf{Out}_{\alpha}(op')$. 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 $\mathsf{Out}_{\gamma_1}(op_l) = \mathsf{Out}_{\alpha}(op_l)$, it follows that $\mathsf{Out}_{\alpha}(op') \prec \mathsf{Out}_{\alpha}(op_l)$. 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 $\mu_{i,b}^{(1)} \leq 2$ in this case.

Thus, in all cases, $\mu_{i,b}^{(1)} \leq 2$, 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 5.3 for the particular case of process p_k is now immediate:

Property 5.4 $1 \leq \mu_{k,b}^{(1)} \leq 2$

By Property 3.2, $\mathsf{Out}_{\gamma_1}(op_l)$ is a composite expression involving for each process p_i, $1 \leq i \leq n$, $\mu_i^{(1)}$ 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

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

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, $\mathsf{Out}_{\alpha}(op_l) = \mathsf{Out}_{\gamma_2}(op_l)$. Take now the (unique) valid serialization $S(\gamma_2)$ of γ_2.

For each process p_i, where $1 \leq i \leq n$, 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 5.5 *For each process p_i, where $1 \leq i \leq n$, $\mu_{i,b}^{(2)} \leq 1$.*

Proof: Consider the *earliest* (if any) operation op at process p_i such that $op \stackrel{S(\gamma_2)}{\longrightarrow} op_l$, 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 $\mu'_{i,b} \leq 1$, as needed. ∎

By Property 3.2, $\mathsf{Out}_{\gamma_2}(op_l)$ is a composite expression involving for each process p_i, $1 \leq i \leq n$, $\mu_i^{(2)} = \mu_{i,a}^{(2)} + \mu_{i,b}^{(2)}$ 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

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

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

Proof: We will prove that both $\mu^{(1)}_{i,a} \leq \mu^{(2)}_{i,a}$ and $\mu^{(2)}_{i,a} \leq \mu^{(1)}_{i,a}$

1. To prove that $\mu^{(1)}_{i,a} \leq \mu^{(2)}_{i,a}$, 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 $\mu^{(1)}_{i,a} \leq \mu^{(2)}_{i,a}$.

2. To prove that, $\mu^{(1)}_{i,a} \leq \mu^{(2)}_{i,a}$, 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 $\mu^{(1)}_{i,a} \leq \mu^{(2)}_{i,a}$.

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

We continue with the second property:

Property 5.7 $\mu^{(1)}_{k,b} - \mu^{(2)}_{k,b} \geq 1$

Proof: Recall from Property 5.4 that $1 \leq \mu^{(1)}_{k,b} \leq 2$. 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, $\mathsf{Out}_{\alpha}(op_k) \prec \mathsf{Out}_{\alpha}(op'_k)$, $op'_k \xrightarrow{\alpha} op_k$, 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 $\mathsf{Out}_{\alpha}(op_k) \prec \mathsf{Out}_{\alpha}(op_l)$. A contradiction.

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

2. Assume now that $\mu^{(1)}_{k,b} = 2$. By Property 5.5, $\mu^{(2)}_{k,b} \leq 1$, so that $\mu^{(1)}_{k,b} - \mu^{(2)}_{k,b} \geq 1$ in this case.

Thus, in all cases, $\mu^{(1)}_{k,b} - \mu^{(2)}_{k,b} \geq 1$, as needed. ■

Since $\mathsf{Out}_{\gamma_1}(op_l) = \mathsf{Out}_{\gamma_2}(op_l)$, 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 2.2, it follows that for each process p_i, where $1 \leq i \leq n$,

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

It follows that

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

Hence, Property 2.1 implies that

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

Consider any index i, where $1 \leq i \leq n$. By Property 5.6, $\mu_i^{(1)} - \mu_i^{(2)} = \mu_{i,b}^{(1)} - \mu_{i,b}^{(2)}$. Now, Properties 5.3 and 5.5 immediately imply that $-1 \leq \mu_{i,b}^{(1)} - \mu_{i,b}^{(2)} \leq 2$, On the other hand, Property 5.7 implies that $\mu_{k,b}^{(1)} - \mu_{k,b}^{(2)} \geq 1$, so that not all differences $\mu_{i,b}^{(1)} - \mu_{i,b}^{(2)}$, where $1 \leq i \leq n$, 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-Sized Switching Networks

In this section, we present our impossibility results for finite-sized switching networks that implement monotone groups.

6.1 Finite-Switch Networks

We first consider switching networks with finite switches. We show:

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

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 $(or(\mathcal{N})+1)\cdot\left(S^{size(\mathcal{N})}+1\right)$ tokens, whose types are $(or(\mathcal{N})+1)\cdot\left(S^{size(\mathcal{N})}+1\right)$-wise independent over $\langle \mathbb{I}\Gamma, \oplus, \rangle$. By the *Monotone Linearizability Lemma* (Proposition 5.2), execution α is linearizable. Write $\alpha = \alpha_1\alpha_2\ldots\alpha_{\lceil or(\mathcal{N})+\frac{1}{2}\rceil}$, where each execution fragment α_i, $1\le i\le \left\lceil or(\mathcal{N})+\frac{1}{2}\right\rceil$, includes the traversals of $S^{size(\mathcal{N})}+1$ tokens.

Take now any execution fragment α_i, where $1\le i\le (or(\mathcal{N})+1)$. Since each token traverses at least one switch, α_i contains at least $S^{size(\mathcal{N})}+1$ configurations; so, it contains at least $S^{size(\mathcal{N})}+1$ internal configurations. Since the total number of internal configurations of $\mathcal{N}$ is $S^{size(\mathcal{N})}$, 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 $1\le i\le 3$. 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 $\left\lceil or(\mathcal{N}) + \frac{1}{2} \right\rceil \cdot \left(S^{size(\mathcal{N})} + 1\right)$-wise independent over $\langle \mathbb{I}\Gamma, \oplus, \rangle$. By the *Monotone Linearizability Lemma* (Proposition 5.2), 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 sequences the operations (that correspond to the tokens) so that they can be atomically invoked (by the tokens) on the register.

Recall from Example 2.1 the *Integers with Addition* monotone group $\langle \mathrm{Z}, \mathbb{N} \setminus \{0\}, +, \leq \rangle$, which is associated with the monotone Fetch&Add operation. So, Theorem 6.1 immediately implies:

Corollary 6.2 *There is no non-trivial, finite-switch switching network $\mathcal{N}$ with concurrency $(or(\mathcal{N}) + 1) \cdot \left(S^{size(\mathcal{N})} + 1\right)$ that has finite size, incurs register bottleneck at least 2 and implements the monotone* Fetch&Add *operation.*

Recall from Example 2.2 the *Rationals with Multiplication* monotone group $\langle \mathrm{Z}, \mathbb{N} \setminus \{0\}, +, \leq \rangle$, which is associated with the monotone Fetch&Multiply operation. So, Theorem 6.1 immediately implies:

Corollary 6.3 *There is no non-trivial, finite-switch switching network $\mathcal{N}$ with concurrency $(or(\mathcal{N}) + 1) \cdot \left(S^{size(\mathcal{N})} + 1\right)$ that has finite size, incurs register bottleneck at least 2 and implements the monotone* Fetch&Multiply *operation.*

6.2 Infinite-State Switches

We now turn to switching networks with infinite switches. Clearly, the proof of Theorem 6.1 is not applicable to switching networks with infinite switches, since the number of their possible network configurations is no longer finite. Thus, we need to develop new techniques to handle this case. We show:

Theorem 6.4 (Impossibility Result for Infinite-Switch Networks) *There exists no non-trivial, infinite-switch switching network with unbounded concurrency that has finite size, incurs layer bottleneck at least* 2 *and implements a monotone group* $\langle \mathbb{I}, \mathbb{M}, \oplus, \preceq \rangle$.

Proof: Assume, by way of contradiction that there is such a switching network $\mathcal{N}$. Partition $\mathcal{N}$ into layers $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_{d(\mathcal{N})}$ in the natural way. Assume, without loss of generality, that any switch b at layer ℓ, where $2 \leq \ell < d(\mathcal{N})$, has its input wires connected to switches of layer $\ell - 1$ and its output wires connected to switches of layer $\ell + 1$.[§]

We will construct a (non-sequential) execution α of network $\mathcal{N}$ involving an unbounded number of tokens $t_1, t_2, \ldots$. issued by distinct processes. In order to prepare the reader for the formal construction of execution α, we provide an informal outline of some details of the construction:

- Token t_1 is the first token which traverses the network $\mathcal{N}$.
- Each subsequent token t_i, where $i > 1$, enters the network when one of the following two, mutually exclusive events occurs to the immediately preceding token t_{i-1}: either token t_{i-1} is exiting the network $\mathcal{N}$ from one of the network's output wires, or token t_i is "halted" on some particular switch without taking the corresponding switch transition. The construction will determine which of the two events will occur for each token.

We now continue with the details of the formal proof.

Let b_1^1 and b_2^1 denote two distinct switches at the first layer ℓ_1 (the existence of these two distinct switches follows from the fact that the layer bottleneck is at least 2 and the number of tokens in execution α are infinite). In execution α, tokens with odd index enter the network $\mathcal{N}$ from switch b_1^1 (*class-1* tokens), and tokens with even index enter the network from switch b_2^1 (*class-2* tokens). Note that there is an infinite number of *class-1* and *class-2* tokens. In execution α we let each token traverse its respective switch of the first layer. Since the number of tokens of *class-1* is infinite, there is a switch b_1^2 at layer ℓ_2 such that an infinite number of tokens of *class-1* access b_1^2. Similarly, there is a switch b_2^2 at layer ℓ_2 such that an infinite number of tokens of *class-2* access b_2^2. We choose the switches b_1^2 and b_2^2 so that are not the same, namely, $b_1^2 \neq b_2^2$. Note that we can always find such distinct switches at layer ℓ_2, since otherwise, there would be only one switch at layer ℓ_2 accessed by an infinite number of tokens from both classes, which implies that there is an infinite suffix of α that violates the assumption that the layer bottleneck is at least 2.

[§]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 missintg layers, with input and output width 1, which simply forward tokens (without routing them).

In execution α, we let all *class-1* tokens that access switch b_1^2 to take their transitions on this switch, and the rest of the *class-1* tokens to halt on their layer ℓ_2 switch (which is different than b_1^2). We treat similarly the *class-2* tokens on switch b_2^2. Note that there is an infinite number of *class-1* and *class-2* tokens that exit from the second layer. Execution α continues in the same fashion for each subsequent layer. As a consequence, in execution α an infinite number of *class-1* and *class-2* tokens finish by leaving the network from an output wire. The *class-1* tokens that finish follow the path $b_1^1, b_1^2, \ldots, b_1^d$, while the *class-1* tokens that do not finish are halted on some switch of this path. Similarly, the *class-2* path is $b_2^1, b_2^2, \ldots, b_2^d$. Since $b_1^i \neq b_2^i$, for any i, $1 \leq i \leq d$, we have that in execution α the path of tokens from *class-1* does not intersect with the path of tokens from *class-2*.

Consider now the shortest prefix α' of execution α in which a token t_{i_1} of *class-1* finishes and a token t_{i_2} of *class-2* finishes. Since α' is a prefix of α it must be that the number of tokens in α' is bounded by some number m. We can choose the input values of the first m tokens of prefix α' so that they are m-wise independent.

Next, we construct a bounded execution β from the prefix α', such that in β all the *class-1* tokens enter the network $\mathcal{N}$ before the *class-2* tokens enter the network (within each class, the relative order of tokens is the same as in α'). Notice that in prefix α' and in execution β the tokens behave in exactly the same way, since the paths of the tokens from the different classes do not intersect. Proposition 5.2 implies that execution β is linearizable, since the input values of the tokens are m-wise independent. Therefore, since in execution β the token t_{i_2} enters the network after token t_{i_1} finishes, it must be $\mathsf{Out}_\beta(t_{i_1}) \prec \mathsf{Out}_\beta(t_{i_2})$. Similar to β, we can construct an execution γ where now the *class-2* tokens enter the network before the *class-1* tokens. Execution γ must be linearizable, and since token t_{i_1} enters the network after token t_{i_2} finishes, it must be that $\mathsf{Out}_\gamma(t_{i_2}) \prec \mathsf{Out}_\gamma(t_{i_1})$. Notice that all the tokens in executions α', β and γ behave in exactly the same way, since tokens from *class-1* and *class-2* do not intersect. Therefore, $\mathsf{Out}_\beta(t_{i_1}) = \mathsf{Out}_\gamma(t_{i_1})$ and $\mathsf{Out}_\beta(t_{i_2}) = \mathsf{Out}_\gamma(t_{i_2})$. Subsequently, $\mathsf{Out}_\beta(t_{i_2}) \prec \mathsf{Out}_\beta(t_{i_1})$. A contradiction. ■

We remark that the assumption of a *non-trivial* switching network is essential for Theorem 6.4 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 argument 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 from Example 2.1 the *Integers with Addition* monotone group $\langle \mathbb{Z}, \mathbb{N} \setminus \{0\}, +, \leq \rangle$, which is associated with the monotone Fetch&Add operation. So, Theorem 6.4 immediately implies:

Corollary 6.5 *There is no non-trivial switching network with infinite switches that has finite size, incurs layer bottleneck at least* 2 *and implements the monotone* Fetch&Add *operation, when the concurrency is unbounded.*

Recall from Example 2.2 the *Rationals with Multiplication* monotone group $\langle \mathbb{Z}, \mathbb{N} \setminus \{0\}, +, \leq \rangle$, which is associated with the monotone Fetch&Multiply operation. So, Theorem 6.4 immediately implies:

Corollary 6.6 *There is no non-trivial switching network with infinite switches that has finite size, incurs layer bottleneck at least* 2 *and implements the monotone* Fetch&Multiply *operation, when the concurrency is unbounded.*

7 The Impossibility of Low-Latency Switching Networks

In this section, we present our impossibility results for low-latency switching networks that implement monotone groups. We show:

Theorem 7.1 (Lower Bound on Latency of Switching Networks) *Take any switching network $\mathcal{N}$ that implements a monotone group $\langle \mathbb{T}, \mathbb{M}, \oplus, \preceq \rangle$. Then, there is a sequential execution of $\mathcal{N}$ with n tokens such that each token traverses at least $\left\lceil \frac{n-1}{c-1} \right\rceil$ switches.*

Proof: Consider n tokens $t_1, t_2, \ldots, t_n$ issued by distinct processes, with respective input values $a_1, a_2, \ldots, a_n \in \mathbb{M}$ which are n-wise independent in $\langle \mathbb{M}, \oplus \rangle$.

By Lemma 4.2, for any arbitrary configuration Q, the preferred paths of any two tokens with respect to configuration Q intersect each other. By the definition of c, no more than $c-1$ tokens (other than t_i) can access any switch along the preferred path of t_i (starting from total configuration s). Since every other process's preferred path must intersect t_i's preferred path, it follows that the preferred path of t_i must include at least $\left\lceil \frac{n-1}{c-1} \right\rceil$ switches.

Take now any sequential execution α of $\mathcal{N}$, starting from any arbitrary quiescent total configuration s, in which each token t_i issues only operations with input value a_i. Since the preferred path of any token includes at least $\left\lceil \frac{n-1}{c-1} \right\rceil$ switches, the first token to run will traverse at least $\left\lceil \frac{n-1}{c-1} \right\rceil$ switches of $\mathcal{N}$, and the network will return to another quiescent total configuration, for which the same argument applies inductively. ■

Recall from Example 2.1 the *Integers with Addition* monotone group $\langle \mathrm{Z}, \mathbb{N} \setminus \{0\}, +, \leq \rangle$, which is associated with the monotone Fetch&Add operation. So, Theorem 7.1 immediately implies:

Corollary 7.2 *For any switching network $\mathcal{N}$ that implements the monotone* Fetch&Add *operation, there is a sequential execution with n tokens such that each token traverses at least $\left\lceil \frac{n-1}{c-1} \right\rceil$ switches.*

Recall from Example 2.2 the *Rationals with Multiplication* monotone group $\langle \mathrm{Z}, \mathbb{N} \setminus \{0\}, +, \leq \rangle$, which is associated with the monotone Fetch&Multiply operation. So, Theorem 7.1 immediately implies:

Corollary 7.3 *For any switching network $\mathcal{N}$ that implements the monotone* Fetch&Multiply *operation, there is a sequential execution with n tokens such that each token traverses at least $\left\lceil \frac{n-1}{c-1} \right\rceil$ switches.*

8 Conclusion and Open Problems

8.1 Review of Results

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 succeded 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 and latency. These negative end results establish the *first* space complexity and time complexity separations between Fetch&Increment and any monotone RMW operation in the model of switching networks.

8.2 Some Remarks

We remark that the proof of Theorem 6.4 requires unbounded (finite or infinite) concurrency. This stems from the assumption that the layer bottleneck is at least 2; thus, in any infinite suffix of an infinite execution, at least two switches in each layer are certain to be accessed by tokens. We do not know, however, how many tokens we will need (and, therefore, how many processes) before a second switch is accessed; hence, we cannot bound the concurrency needed in the execution used in the proof. This was not the case for switching networks with finite switches (even though the assumption on register bottleneck made there is quite similar), where the fact that configurations were repeated (since switches are finite) allowed to get a token visit a register for a second time by shepherding in a finite number of tokens. So, since the proof of Theorem 6.4 requires unbounded concurrency, it does *not* imply Theorem 6.1 which assumes bounded concurrency, and the two results are incomparable and represent a trade-off.

8.3 Directions for Further Research

We conclude this article with a collection of problems that remain tantalizingly open.

1. On the most concrete level, what is the *least* lower bound on concurrency that suffices to prove the impossibility of a (non-trivial) concurrent, low-contention switching network with finite switches implementing an arbitrary monotone RMW operation? The proof of Theorem 6.1 implies that this least lower bound is no more than $3\, or(\mathcal{N})\, S^{s(\mathcal{N})}$. Can this lower bound be further reduced?

2. Still on the most concrete level, we do not know yet whether the convenience of unbounded concurrency employed in the proof of Theorem 6.4 is necessary, or whether there is instead a (non-trivial) concurrent, low-contention switching network with infinite switches that implements an arbitrary monotone RMW operation in the presence of bounded concurrency.

3. The relative power of finite switches and infinite switches deserves further study. For example, is there some (not necessarily monotone) RMW operation that is implementable by highly concurrent, low-contention, finite-sized switching networks with infinite switches, but *not* by such ones with finite switches?

4. The possibility of implementing other, *non-monotone* RMW operations with finite-sized, highly concurrent, low-contention switching networks remains open. For example, does the Swap operation enjoy such implementations?

5. Which (if any) *timing conditions* would suffice to overcome the impossibility results we have shown? Recall that timing conditions have been exploited in the work of Lynch *et al.* [27] for devising *finite-size* linearizable counting networks, while Herlihy *et al.* [20] establish that no finite-size (non-trivial) *asynchronous* linearizable counting network exists.

6. What are (if any) the limitations on various kinds of distributed systems (other than switching networks) that implement a monotone RMW operation?

A natural candidate to consider is the message-passing system adopted in the work by Wattenhofer and Widmayer [34]; that work showed a lower bound on the message complexity of implementing the Fetch&Increment operation in that system; we feel that similar limitations hold for implementations of any monotone RMW operation. Other candidates include the *transactional* model studied by Tuzhilin and Spirakis [33], the model of *counting protocols* due to Moran and Taubenfeld [31], and the recent *peer-to-peer* model due to Giesen *et al.* [16].

Identifying appropriate space and time complexity-theoretic measures for these models, and proving bounds on the measures for concurrent, low-contention implementations of (monotone) RMW operations in these models, is a wide avenue for further research.

7. Although our work has established that appropriate generalizations of counting networks *cannot* provide distributed data structures for the RMW operation that simultaneously achieve finite size, low latency and low contention, the existence of some other distributed data structures with these properties is not excluded yet. However, our work implies that such structures will require a completely new approach.

Acknowledgements

We would like to thank the anonymous SIROCCO 2003 reviewers for their helpful comments.

References

[1] E. Aharonson and H. Attiya, "Counting Networks with Arbitrary Fan-out," *Distributed Computing,* Vol. 8, No. 4, pp. 163–169, 1995.

[2] W. Aiello, C. Busch, M. Herlihy, M. Mavronicolas, N. Shavit and D. Touitou, "Supporting Increment and Decrement Operations in Balancing Networks," *Chicago Journal of Theoretical Computer Science,* 2000-4, December 14, 2000 (electronic).

Preliminary version appeared in *Proceedings of the 16th International Symposium on Theoretical Aspects of Computer Science,* G. Meinel and S. Tison eds., pp. 377–386, Vol. 1563, Lecture Notes in Computer Science, Springer-Verlag, Trier, Germany, March 1999.

[3] J. Aspnes, M. Herlihy and N. Shavit, "Counting Networks," *Journal of the ACM,* Vol. 41, No. 5, pp. 1020–1048, September 1994.

Preliminary version appeared as "Counting Networks and Multi-Processor Coordination," in *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing,* pp. 348–358, May 1991.

[4] H. Attiya and J. L. Welch, "Sequential Consistency versus Linearizability," *ACM Transactions on Computer Systems,* Vol. 12, No. 2, pp. 91–122, May 1994.

[5] H. Attiya and J. L. Welch, *Distributed Computing: Fundamentals, Simulations and Advanced Topics,* MvGraw-Hill, 1998.

[6] K. E. Batcher, "Sorting Networks and their Applications," *Proceedings of AFIPS Spring Joint Computer Conference,* Vol. 32, pp. 307–314, 1968.

[7] J. E. Burns and N. A. Lynch, "Bounds on Shared Memory for Mutual Exclusion," *Information and Computation,* Vol. 107, No. 2, pp. 171–184, December 1993.

[8] C. Busch, *A Study on Distributed Structures,* Ph. D. Thesis, Department of Computer Science, Brown University, May 2000.

[9] C. Busch and M. Herlihy, "Sorting and Counting Networks of Arbitrary Width and Small Depth," *Theory of Computing Systems,* Vol. 35, No. 2, pp. 99–128, January 2002.

[10] C. Busch and M. Mavronicolas, "A Combinatorial Treatment of Balancing Networks," *Journal of the ACM,* Vol. 43, No. 5, pp. 794–839, September 1996.

[11] C. Busch and M. Mavronicolas, "An Efficient Counting Network," *Proceedings of the 1st Merged International Parallel Processing Symposium and IEEE Symposium on Parallel and Distributed Processing,* pp. 380–385, May 1998.

[12] M. Dowd, Y. Perl, L. Rudolph and M. Saks, "The Periodic Balanced Sorting Network," *Journal of the ACM,* Vol. 36, No. 4, pp. 738–757, October 1989.

[13] C. Dwork, M. Herlihy and O. Waarts, "Contention in Shared Memory Algorithms," *Journal of the ACM,* Vol. 44, No. 6, pp. 779–805, November 1997.

[14] K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM,* Vol. 19, No. 11, pp. 624–633, November 1976.

[15] P. Fatourou and M. Herlihy, "Adding Networks," *Proceedings of the 15th International Symposium on DIStributed Computing,* J. L. Welch ed., pp. 330–342, Vol. 2180, Lecture Notes in Computer Science, Springer-Verlag, Lisbon, Portugal, October 2001.

[16] J. Giesen, R. Wattenhofer and A. Zollinger, "Towards a Theory of Peer-to-Peer Computability," *Proceedins of the 9th International Colloquium on Structural Information and Communication Complexity,* pp. 115–132, June 2002.

[17] A. M. W. Glass, *Partially Ordered Groups,* Vol. 7, Series in Algebra, World Scientific, 1999.

[18] J. Goodman, M. Vernon and P. Woest, "Efficient Synchronization Primitives for Large-Scale, Cache-Coherent Multiprocessors," *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems,* pp. 64–75, April 1989.

[19] M. Hall, Jr., *Theory of Groups,* Second Edition, Chelsea Publishing Company, 1979.

[20] M. Herlihy, N. Shavit and O. Waarts, "Linearizable Counting Networks," *Distributed Computing,* Vol. 9, No. 4, pp. 193–203, February 1996.

[21] M. Herlihy, S. Tirthapura and R. Wattenhofer, "Ordered Multicast and Distributed Swap," *Operating Systems Review,* Vol. 35, No. 1, pp. 85–96, January 2001.

[22] M. Herlihy and J. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Transactions on Programming Languages and Systems,* Vol. 12, No. 3, pp. 463–492, July 1990.

[23] M. Klugerman and C. G. Plaxton, "Small-Depth Counting Networks," *Proceedings of the 24th Annual ACM Symposium on Theory of Computing,* pp. 417–428, May 1992.

[24] C. P. Kruskal, L. Rudolph and M. Snir, "Efficient Synchronization on Multiprocessors with Shared Memory," *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing,* pp. 218–228, August 1986.

[25] L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers,* Vol. C-28, No. 9, pp. 690–691, September 1979.

[26] N. Lynch, *Distributed Algorithms,* Morgan Kaufmann, 1996.

[27] N. Lynch, N. Shavit, A. Shvartsman and D. Touitou, "Timing Conditions for Linearizability in Uniform Counting Networks," *Theoretical Computer Science,* Vol. 220, No. 1, pp. 67–91, June 1999.

[28] N. Lynch and M. Tuttle, "An Introduction to Input/Output Automata" *CWI Quarterly,* Vol. 2, No. 3, pp. 219–246, September 1989.

[29] M. Mavronicolas, M. Merritt and G. Taubenfeld, "Sequentially Consistent versus Linearizable Counting Networks," *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing,* pp. 133–142, May 1999.

[30] J. Misra, "Axioms for Memory Access in Asynchronous Hardware Systems," *ACM Transactions on Programming Languages and Systems,* Vol. 8, No. 1, pp. 142–153, January 1986.

[31] S. Moran and G. Taubenfeld, "A Lower Bound on Wait-Free Counting," *Journal of Algorithms,* Vol. 24, No. 1, pp. 1–19, July 1997.

[32] N. Shavit and D. Touitou, "Elimination Trees and the Construction of Pools and Stacks," *Theory of Computing Systems,* Vol. 30, No. 6, pp. 545–570, November/December 1997.

[33] A. Tuzhilin and P. Spirakis, "A Semantic Approach to Correctness of Concurrent Transaction Executions," *Proceedings of the 4th Annual ACM Symposium on Principles of Database Systems,* pp. 85–96, 1985,

[34] R. Wattenhofer and P. Widmayer, "An Inherent Bottleneck in Distributed Counting," *Journal of Parallel and Distributed Computing,* Vol. 49, pp. 135–145, February 1998.