# Read-Modify-Write Networks [*]

*Panagiota Fatourou*[†] *Maurice Herlihy*[‡]

**Abstract**

A *read-modify-write* register for a set of functions $F$ provides an operation that atomically (1) returns the variable's current value $v$, and (2) replaces that value with $f(v)$, where $f$ is a function in $F$.

A *read-modify-write network* is a distributed data structure that implements a concurrent, lock-free, low-contention read-modify-write register. For example, counting networks ([3]) are a family of read-modify-write networks that support atomic increments and decrements.

We consider the problem of constructing read-modify-write networks for particular sets of functions. Of particular interest are the read-modify-write networks `fetch&add`, which add to the value an arbitrary number, and `fetch&mul`, which multiply the value by an arbitrary number.

We identify a simple algebraic property of the function set $F$ that requires any distributed read-modify-write implementation for $F$ to have high latency. Any such network has sequential executions in which each token traverses a number of switching elements essentially linear in the number of processes. By contrast, there exist counting networks ([9, 18]) in which tokens traverse a logarithmic number of switches.

We provide a matching upper bound for a large class of read-modify-write networks including `fetch&add`, `fetch&mul`, and related networks of interest.

[*]A preliminary version of this work appears in the *Proceedings of the 15th International Symposium on DIStributed Computing,* pp. 330-341, Lisbon, Portugal, October 2001. This work also appears as a brief announcement in the *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing,* Newport, Rhode Island, August 2001.

[†]Department of Computer Science, University of Ioannina, P.O. Box 1186, 45110 Ioannina, Greece. The work of this author is partially supported by funds from the Program E.Π.E.A.E.K. II of the Greek Ministry of Education. Part of the work of the author was performed while at the Max-Planck Institut für Informatik, Saarbrücken, Germany, while at the Department of Computer Science, University of Toronto, Canada, and while visiting the Department of Computer Science, Brown University, Providence, USA. E-mail: `faturu@cs.uoi.gr`

[‡]**Contact Author:** Department of Computer Science, Brown University, Providence, P.O. Box 1910, RI 02912, USA. Tel.: 401-863-7646, Fax: 401-863-7657 , Email: `herlihy@cs.brown.edu`

# 1 Introduction

## 1.1 Motivation and Overview

A *read-modify-write* register [19] is characterized by a *domain* $D$ (usually the integers), and a set of functions $F$ from $D$ to $D$. The register `r` provides an operation `r.rmw`$(f)$, where $f$ is a function in $F$. This operation atomically (1) returns the register's current value $v$ and (2) replaces that value with $f(v)$. Many well-known synchronization primitives can be expressed as read-modify-write variables. For example, a `fetch&inc` register provides an operation that atomically adds one to its value and returns its prior value. Here $F$, the set of functions, is a singleton set containing only $\oplus_1(x) = x+1$. Applications of `fetch&inc` counters include shared pools and stacks, load balancing, and software barriers. A `fetch&add` register provides an operation that adds an arbitrary number to the value. Here, the set of functions $F$ is the set $\{\oplus_a \mid \oplus_a(x) = x + a, \forall a \in \mathbb{Z}\}$, where $\mathbb{Z}$ is the set of integers. A `fetch&mul` register does the same for multiplication. In principle, one could define read-modify-write registers over domains other than the integers, encompassing, for example, sequences or matrix multiplication.

A *counting network* [3] is a class of distributed data structures used to construct concurrent, low-contention implementations of read-modify-write registers that can be incremented or decremented [2, 24]. Here, the function set has two elements: $\oplus_1(x) = x + 1$ and $\ominus_1(x) = x - 1$.

It is natural to ask whether counting networks can be generalized to support lock-free, highly-concurrent, low-contention implementations of other kinds of read-modify-write registers. In this paper we show that if the set of functions $F$ contains at least two elements satisfying a simple algebraic property, then any network implementation of such a read-modify-write register has inherently high latency, even in *sequential* executions in which processes traverse the network one completely after the other. Both `fetch&add` and `fetch&mul` registers are subject to this lower bound.

## 1.2 Functional Definitions

Let $f : D \to D$ be a function. Let $f^k$ denote the $k$-fold composition of $f$ (where $f^0 = \mathbf{1}$, the identity function); for example, $f^2(v) = f(f(v))$. Fix any integer $n > 0$. Let the $n$-*orbit* of $v$ under $f$ be the set

$$\left\{v, f(v), f^2(v), \ldots, f^{n-1}(v)\right\}.$$

An $n$-orbit is *aperiodic* if, for all $v$ in $D$, $f^i(v) \neq f^j(v)$ for $0 \leq i \neq j < n$. We say that function $f$ *displaces* function $g$, if, for all $v$ in $D$,

- the $n$-orbit of $v$ under $g$ is aperiodic, and
- the $n$-orbits of $v$ and $f(v)$ under $g$ are disjoint.

Schematically, displacement can be illustrated by the following diagram, where all expressions are intended to be distinct.

$$\begin{tikzcd}
v \arrow[r] \arrow[d] & g(v) \arrow[r] & \cdots \arrow[r] & g^{n-1}(v) \\
f(v) \arrow[r] & g(f(v)) \arrow[r] & \cdots \arrow[r] & g^{n-1}(f(v))
\end{tikzcd}$$

Let us review some examples of functions that displace one another. Fix any arbitrary integers $a$ and $b$.

**Integer Addition** Define $\oplus_a(v) = v + a$. We claim that if $b$ does not divide $a$, then $\oplus_a$ displaces $\oplus_b$. Clearly, $\oplus_b$ has aperiodic $n$-orbits. For every $x$ in the $n$-orbit of $v$ under $\oplus_b$, $x \equiv v \bmod b$, and for every $y$ in the $n$-orbit of $v + a$ under $\oplus_b$, $y \equiv v + a \bmod b$, which are distinct because $b$ does not divide $a$.

**Integer Multiplication** Define $\otimes_a(x) = a \cdot x$. We claim that if $a$ is not a power of $b$, then $\otimes_a$ displaces $\otimes_b$. Clearly, $\otimes_b$ has aperiodic $n$-orbits. Every $x$ in the $n$-orbit of $v$ under $\otimes_b$ has the form $b^i \cdot v$, while every $x$ in the $n$-orbit of $\otimes_a(v)$ under $\otimes_b$ has the form $b^i \cdot a \cdot v$. These orbits are distinct because $a$ is not a power of $b$.

**Multiset Union** Let $D$ be an arbitrary set, and $S$ a multiset of elements of $D$. Define $\uplus_a(S) = S \uplus \{a\}$, where "$\uplus$" denotes multiset union. We claim that if $a$ and $b$ are distinct, then $\uplus_a$ displaces $\uplus_b$. Clearly, $\uplus_b$ has aperiodic $n$-orbits. Every $S'$ in the $n$-orbit of $S$ under $\uplus_b$ consists of $S$ and some number of copies of $b$, while every $S''$ in the $n$-orbit of $\uplus_a(S)$ under $\uplus_b$ consists of $S$, some number of copies of $b$, and an additional copy of $a$.

**Sequences** If $\sigma$ is a sequence of integers, define $\odot_a(S) = S \cdot a$, where "$\cdot$" denotes concatenation. We claim that if $a$ and $b$ are distinct, then $\odot_a$ displaces $\odot_b$. Clearly, $\odot_b$ has aperiodic $n$-orbits. Every $\sigma'$ in the $n$-orbit of $S$ under $\odot_b$ consists of $\sigma$ followed by some number of copies of $b$, while every $\sigma''$ in the $n$-orbit of $\odot_a(S)$ under $\odot_b$ consists of $\sigma \cdot a$, followed by some number of copies of $b$.

Similar examples exist for functions over other domains, such as vector addition, matrix multiplication, and so on.

## 1.3 Summary of Switching Networks

Informally, a *switching network* is a generalization of a balancing network used to construct counting networks. As discussed in detail in Section 2, a switching network is a directed graph, where edges are called *wires* and nodes are called *switches*. Each of the $n$ processes shepherds a *token* through the network. Switches and tokens are allowed to have internal states. A token arrives at a switch via an input wire. In one atomic step, the switch absorbs the token, changes its state and possibly the token's state, and emits the token on an output wire.

Note that switching networks are more powerful than balancing networks, since each switch can have arbitrary state (instead of a single bit) and tokens also have state. Since most of this paper concerns lower bounds, it is sensible to examine the most powerful (natural) generalization of balancing networks.

Let $\mathcal{R}$ be a switching network. The *one-shot contention* of $\mathcal{R}$ is the largest number of tokens that can meet at a single switch in any execution in which exactly $n$ tokens enter $\mathcal{R}$ on distinct wires. It is natural to define a low-contention switching network to be one with constant one-shot contention. Indeed, it is not difficult to prove (Section 2) that the one-shot contention of counting networks, as well as of a large class of switching networks (including the read-modify-write network presented in Section 4) is constant. The *worst-case contention* of a switching network $\mathcal{R}$ is the largest number of tokens that can meet at a single switch in any execution of $\mathcal{R}$.

## 1.4 Contribution and Significance

The principal contribution of this paper is to show that counting networks cannot easily be generalized to support larger classes of operations. Switching networks, the natural generalization of counting networks, cannot provide satisfactory support even for such simple operations as addition or multiplication. Counting networks are practical because they have both low contention and latency logarithmic in the number of concurrent processes, yielding real parallelism. By contrast, any switching network implementation of `fetch&add`, `fetch&mul`, or related read-modify-write registers that has low contention must have latency linear in the number of concurrent processes, thus yielding little or no parallelism.

Specifically, we show that if the family $F$ contains functions $f$ and $g$ such that $f$ displaces $g$, then in *any sequential execution* of any read-modify-write network implementing $F$, all tokens applying function $f$ traverse $\Omega(n/c)$

switches, where $n$ is the number of processes and $c$ is the network's one-shot contention (typically a small constant). This lower bound holds for all switching networks whose set of functions satisfies the displacement property, independently of the topology of the network or the state of switches and tokens Moreover, it holds for all sequential executions independently of the number of tokens involved in the execution. Aspnes *et al.* [3] have proved that the worst-case contention of any counting network is $\Theta(n)$[1]. We remark that our $\Omega(n/c)$ lower bound is expressed in terms of the one-shot (and not the worst-case) contention, which implies that any low-contention read-modify-write network must have high worst-case latency, even in the absence of concurrency.

As an aside, we note that there are several interesting cases of read-modify-write registers not subject to our lower bound. If $F$ is a singleton set $\{f\}$, or if $F = \left\{f, f^{-1}\right\}$ is a set consisting of $f$ and its inverse, then the read-modify-write register can be implemented by a counting network (extended to support decrements). The process obtaining the $i$th value takes $f^i(x_0)$, where $x_0$ is the initial value of the register. If we augment $F$ with the identity function, then the resulting read-modify-write register is just a regular counting network augmented by a pure read operation. Other read-modify-write registers, such as `swap`, remain the focus of ongoing research [16].

We give a matching upper bound for an important and useful class of read-modify-write registers that encompasses `fetch&add` and `fetch&mul`. A set of functions $F$ is *commutative* if, for every $f$ and $g$ in $F$, and every value $v$ in the domain $D$, $f(g(v)) = g(f(v))$. We present a lock-free, low-contention read-modify-write switching network, called LADDER, that implements a read-modify-write register for any commutative $F$. In any sequential execution of LADDER all tokens traverse $O(n)$ switches, while in the presence of concurrency tokens traverse $O(n)$ switches on average. The construction is lock-free, but not wait-free (meaning that individual tokens can be overtaken arbitrarily often, but that some tokens will always emerge from the network in a finite number of steps). LADDER is the *first* concurrent, lock-free, low-contention networked data structure that supports arbitrary commutative read-modify-write operations.

An ideal switching network (like an ideal counting network [15]) is (1) lock-free, with (2) low contention, and (3) low latency. Although this paper

[1]However, in order for high contention to be created, a large number of tokens must first traverse the network at relatively little contention (yielding a low amortized contention) [10].

shows that no switching network can have all three properties, any two are possible:[2] a single switch is lock-free with low latency, but has high contention, a combining network [12] has low contention and $O(\log n)$ latency but requires tokens to wait for one another, and the LADDER construction introduced here is lock-free with low contention, but has $\Omega(n)$ latency.

## 1.5 Related Work

Counting networks were first introduced by Aspnes *et al.* [3]. A flurry of research on counting networks followed (see e.g., [1, 2, 7, 8, 9, 10, 15, 18, 20, 22, 24, 26]). Counting networks are limited to support only `fetch&inc` and `fetch&dec` operations. We generalize traditional counting networks by introducing switching networks, which employ more powerful switches and tokens. Our work is the first to study whether lock-free network data structures can support even more complex operations.

A counting network is *linearizable* [17] if no token $t$ entering the network after some other token $t'$ has exited the network takes a value larger than the one taken by $t'$. Herlihy *et al.* [15] prove that there exists no ideal linearizable counting network. It is easy to show that any network that supports, for example, addition by two distinct integers $a$ and $b$, where $|a| > |b| > 0$, has linearizable executions.

The LADDER read-modify-write network has the same topology as SKEW, the linearizable counting network presented in [15], but the behavior of LADDER is significantly different. In this network, tokens accumulate state as they traverse the network, and they use that state to determine how they interact with switches. The resulting network is substantially more powerful, and requires a substantially different analysis.

Apart from counting networks, a number of techniques and research results for distributed counting have been presented in the literature (for example, see [4, 12, 23, 27, 28, 29]). None of these techniques achieves all characteristics of ideal counting, namely, lock-freedom, low contention, and low latency. Herlihy *et al.* provide in [14] an experimental investigation of the scalability of a variety of software counting techniques. Comprehensive surveys on distributed counting are provided in [5, 6, 27].

Counting networks have been extended in a number of ways. *Diffracting trees* [26], *elimination pools* [24], and *combining funnels* [25] reduce contention by randomization and by exploiting the LIFO nature of certain stack-like data structures to "short-circuit" certain operations. It remains

[2]NASA's erstwhile motto "faster, cheaper, better" has been satirized as "faster, cheaper, better: pick any two".

to be seen whether techniques intended to reduce contention can be adapted to extend functionality.

Giesen *et al.* [11] have introduced a novel alternative approach to concurrent data structures in which processes are not restricted to a fixed switching network. Instead, they are restricted in that they can only access the read-modify-write registers that they share with their neighbors. Giesen *et al.* prove that tasks such as leader election, or deriving a total ordering among the processes such that each process knows its predecessor cannot be implemented in a decentralized way. Unifying this model with the switching network model is an open research area.

### 1.6 Organization

This paper is organized as follows. Section 2 introduces switching networks. Section 3 presents our lower bound, while Section 4 introduces and analyzes the Ladder network. Section 5 concludes with a further discussion of our results and some open problems.

## 2 Switching Networks

### 2.1 Description

A *switching network*, like a counting network [3], is a directed graph whose nodes are simple computing elements called *switches*, and whose edges are called *wires.* A wire directed from switch $s$ to switch $s'$ is an *output wire* for $s$ and an *input wire* for $s'$. The network's *input wires* are those input wires not linked to an output, and similarly for the network's *output wires.* A $(w_{in}, w_{out})$-switching network has $w_{in}$ input wires and $w_{out}$ output wires. The *width* of the network, denoted $w$, is defined to be $w = \max\{w_{in}, w_{out}\}$. A *switch* is a shared data object characterized by an internal state, its set of $f_{in}$ *input wires*, labeled $0, \ldots, f_{in}-1$, and its set of $f_{out}$ *output wires*, labeled $0, \ldots, f_{out}-1$. The values $f_{in}$ and $f_{out}$ are called the switch's *fan-in* and *fan-out* respectively. A *binary switch* is a switch with only two input and two output wires. Figure 1(a) illustrates a binary switch, while Figure 1(b) shows a switching network of width 4.

There are $n$ processes that move (*shepherd*) *tokens* through the network. Each process enters its token on one of the network's $w_{in}$ input wires. After the token has traversed a sequence of switches, it leaves the network on one of its $w_{out}$ output wires. A process shepherds only one token at a time, but it can start shepherding a new token as soon as its previous token has

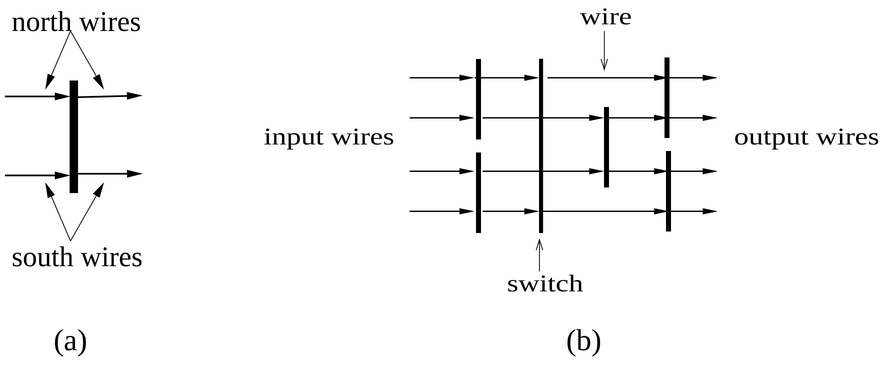

Figure 1: (a) A binary switch. (b) A switching network of width 4.

emerged from the network. Processes work asynchronously, but they do not fail. In contrast to counting networks, each token has a mutable state (a set of fields), which can change as it traverses the network.

A switch acts as a router for tokens. When a token arrives on a switch's input wire, the following events can occur atomically: (1) the switch removes the token from the input wire, (2) the switch changes state, (3) the token changes state, and (4) the switch places the token on an output wire. The wires are one-way communication channels and allow reordering. Communication is asynchronous but reliable (meaning a token does not wait on a wire forever). For each $(f_{in}, f_{out})$-switch, we denote by $x_i$, $0 \leq i \leq f_{in} - 1$, the number of tokens that have entered on input wire $i$, and similarly we denote by $y_j$, $0 \leq j \leq f_{out} - 1$, the number of tokens that have exited on output wire $j$. As an example, a $(k, \ell)$-*balancer* is a switch with fan-in $k$ and fan-out $\ell$ such that the $i$-th input token is routed to output wire $i \bmod \ell$. Counting networks are constructed from balancers and from simple (usually two-input two-output) counting switches.

For any switching network, we denote by $\phi$ the maximum fan-in of any of its switches. For any wire $z$ of the network, the *depth* of $z$, denoted $d(z)$, is defined to be 0 if $z$ is an input wire of the network, and $\max_{i=0,\ldots,f_{in}-1}\{d(x_i)\} + 1$ for an output wire of an $(f_{in}, f_{out})$-switch with input wires $x_i$, $0 \leq i \leq f_{in} - 1$. The *depth* of a switching network, denoted $d$, is the maximal depth of any of its wires. We remark that any path from an input to an output wire of a switching network with depth $d$ traverses at most $d$ switches.

## 2.2 States and Executions

It is convenient to characterize a switch's internal state as a collection of variables, possibly with initial values. A switch's state is given by its internal state and the collection of tokens on its input and output wires. Each token's state is also characterized by a set of variables. Notice that a token's state is part of the state of the process that owns it. A process may change the state of its token while moving it through a switch. A switching network's state is just the collection of the states of its switches.

A switch is *quiescent* if the number of tokens that arrived on its input wires equals the number that have exited on its output wires:

$$\sum_{i=0}^{f_{in}-1} x_i = \sum_{j=0}^{f_{out}-1} y_j.$$

The *safety property* of a switch states that in any state,

$$\sum_{i=0}^{f_{in}-1} x_i \geq \sum_{j=0}^{f_{out}-1} y_j,$$

that is, a switch never creates tokens spontaneously. The *liveness property* states that any switch will eventually become quiescent after handling any finite number of input tokens. A switching network is *quiescent* if all its switches are quiescent.

We denote by $\pi = \langle t, s \rangle$ the *state transition* in which the token $t$ is moved from an input wire to an output wire of a switch $s$. Although transitions can occur concurrently, it is convenient to model them using an interleaving semantics in the style of Lynch and Tuttle [21]. If a token $t$ is on one of the input wires of a switch $s$ at some network state $q$, we say that $t$ is *in front* of $s$ at state $q$ or that the transition $\langle t, s \rangle$ is *enabled* at state $q$. An *execution fragment* $\alpha$ of the network is either a finite sequence $q_0, \pi_1, q_1, \ldots, \pi_n, q_n$ or an infinite sequence $q_0, \pi_1, q_1, \ldots$ of alternating network states and transitions such that for each $\langle q_i, \pi_{i+1}, q_{i+1} \rangle$, the transition $\pi_{i+1}$ is enabled at state $q_i$ and carries the network to state $q_{i+1}$. If $\pi_{i+1} = \langle t, s \rangle$ we say that token $t$ *takes a step* at state $q_i$ (or that $t$ *traverses* $s$ at state $q_i$). An execution fragment beginning with an initial state is called an *execution*. If $\alpha$ is a finite execution fragment of the network and $\alpha'$ is any execution fragment that begins with the last state of $\alpha$, then we write $\alpha \cdot \alpha'$ to represent the sequence obtained by concatenating $\alpha$ and $\alpha'$ and eliminating the duplicate occurrence of the last state of $\alpha$.

For any token $t$, a *$t$-solo execution fragment* is an execution fragment in which only token $t$ takes steps. A *$t$-complete execution fragment* is an execution fragment in which token $t$ exits the network. A finite execution is *complete* if it results in a quiescent state. An execution is *sequential* if for any two transitions $\pi = \langle t, s\rangle$ and $\pi' = \langle t, s'\rangle$, all transitions between them also involve token $t$; that is, tokens traverse the network one completely after the other.

## 2.3 Latency and Contention

The *worst-case latency* of a switching network is the maximum number of switches traversed by any single token in any execution. The *average latency* of a finite execution of a switching network is the total number of switches traversed by all tokens in the execution divided by the number of tokens involved in the execution. The *average latency* of an infinite execution is the limit of the average latency of its finite prefixes. The *average latency* of a switching network is the maximum of the average latency of any of its executions.

The *contention of an execution* is the maximum number of tokens that can be in front of any particular switch at any point during the execution. In a *one-shot* execution, only $n$ tokens (one per process) traverse the network and those tokens are evenly distributed on the input wires (that is, either $\lfloor n/w \rfloor$ or $\lceil n/w \rceil$ tokens enter each input wire). The *one-shot contention* of a switching network, denoted $c$, is the maximum contention over all its one-shot executions. The *worst-case contention* of a switching network is the maximum contention of any of its executions.

The *$\ell$-balancing property* of a switch $s$ requires that if at most $\ell$ tokens reach each input wire of $s$ then at most $\ell$ tokens exit on each of its output wires. We say that a switching network is an *$\ell$-balancing* network if all its switches preserve the $\ell$-balancing property. A straightforward induction argument [15] shows that in any execution $\alpha$ of an $\ell$- balancing network, in which no more than $\ell$ tokens enter on any input wire of the network, there are never more than $\ell$ tokens on any wire of the network. This and the definition of the one-shot contention implies that:

**Lemma 2.1** *Any $\ell$-balancing switching network, where $\ell > 0$, has one-shot contention $O((n/w)\phi)$.*

For any integer $\ell > 0$, a counting network is an $\ell$-balancing network. Therefore:

**Lemma 2.2** *Any counting network has one-shot contention $O((n/w)\phi)$.*

Recall that $w$ is the width of the network, while $\phi$ is the maximum fan-in of any of its switches. We remark that $\phi$ is a small constant (usually equal to two) for all practical counting networks. Moreover, in most implementations of counting networks, the network has $n$ input wires, and each process is preassigned to one of them; that is, $w = n$. Therefore, Lemmas 2.1 and 2.2 imply that the one-shot contention $c$ of a large class of switching networks, including conventional counting networks, is constant. As an example, consider the class of switching networks with $\Omega(n)$ input wires whose switches produce any permutation of their input tokens on their output wires. A straightforward induction argument shows that each switching network of this class has the 1-balancing property, and thus in a one-shot execution, the network never has more than one token on each wire. It follows that the one-shot contention of such a network is bounded by the maximum fan-in of any of its switches.

## Read-Modify-Write Networks

For a domain $D$ and a set of functions $F$, a *read-modify-write network* $\mathcal{R}$ is a switching network that implements a read-modify-write register for $F$. More formally, let $\ell > 0$ be any integer and consider any complete execution $\alpha$ which involves $\ell$ tokens $t_1, \ldots, t_\ell$, where token $t_i$ is labeled with function $f_i$ in $F$. We say that $t_i$ *applies* $f_i$; moreover, we say that $t_i$ *takes* the value $v_i$ that the operation returns in $\alpha$.

Let $v$ be the initial register value. Let $v_i$ be the value taken by token $t_i$ in $\alpha$. The *read-modify-write property* for $\alpha$ states that there exists a permutation $i_1, \ldots, i_\ell$ of $1, \ldots, \ell$, called the *serialization order*, such that (1) $v_{i_1} = v$, and (2) for each $j$, $1 \leq j < \ell$, $v_{i_{j+1}} = f_{i_j}(v_{i_j})$. The first token in the order returns the initial value of the register, and each subsequent token returns the result of applying the functions that precede it. There may be more than one possible serialization orders.

An execution of a read-modify-write network is *linearizable* if for any two tokens $t$ and $t'$ such that $t'$ entered the network after $t$ has exited it, it holds that $v_t < v_{t'}$, where $v_t, v_{t'}$ are the values taken by $t$ and $t'$, respectively. A read-modify-write network is *linearizable* if all its executions are linearizable.

## 3 Lower Bound

Let $\mathcal{R}$ be a read-modify-write network for a set of functions $F$, where $F$ contains $f$ and $g$ such that $f$ displaces $g$. We claim that any token applying $f$ which runs by itself through $\mathcal{R}$, must traverse at least $\lceil (n-1)/(c-1) \rceil$ switches, where $c$ is the one-shot contention of the network.

Start with the network in a quiescent state $q_0$, and assume that the value at $q_0$ of the read-modify-write register is $v$, so the next token to traverse the network in isolation will take value $v$ (independently of the function to be applied by the token on the register). Place token $t$ applying $f$ on one of the input wires. Let $\mathcal{S}$ be the set of switches that $t$ traverses in a $t$-solo, $t$-complete execution fragment starting from $q_0$.

Consider $n-1$ tokens $t_1, \ldots t_{n-1}$, each applying function $g$. Assume that all $n$ tokens $t, t_1, \ldots, t_{n-1}$ are evenly placed on input wires. We construct an execution in which each token $t_i$, $1 \leq i \leq n-1$, must traverse some switch of $\mathcal{S}$.

**Proposition 3.1** *For each $i$, $1 \leq i \leq n-1$, there exists a $t_i$-solo execution fragment $\alpha_i$ with final state $q_i$ starting from state $q_{i-1}$ such that $t_i$ is in front of a switch $s_i \in \mathcal{S}$ at state $q_i$.*

To give the intuition for the proof, consider the special case of an adding network with three processes ($n = 3$), where $f = \oplus_1$ and $g = \oplus_2$; thus, token $t$ applies function $\oplus_1$, while each of the tokens $t_1, t_2$ applies function $\oplus_2$ to the register. Assume, without loss of generality, that $v$ is even. Consider first an execution fragment $\alpha'_1$ starting from $q_0$ in which $t_1$ runs solo all the way through the network. Clearly, $t_1$ takes the even value $v$ in $\alpha'_1$. Assume that $t_1$ never traverses some switch of $\mathcal{S}$ in $\alpha'_1$. Then, if we let $t$ run solo all the way through the network starting from the final state of $\alpha'_1$, it would encounter the same switches in the same internal states as it would starting from $q_0$. Therefore, $t$ takes the even value $v$ as well. However, one of the tokens must take an odd value. These imply that $t_1$ traverses at least one switch of $\mathcal{S}$ in $\alpha'_1$. Take $\alpha_1$ to be the shortest prefix of $\alpha'_1$ such that $t_1$ is in front of a switch of $\mathcal{S}$ at the final state $q_1$ of $\alpha_1$. Clearly, $t_1$ does not traverse any switch of $\mathcal{S}$ in $\alpha_1$.

Consider now an execution fragment $\alpha'_2$ starting from $q_1$ in which $t_2$ runs solo all the way through the network. Since $t$ takes no steps in $\alpha_1 \cdot \alpha'_2$, $t_2$ takes an even value in $\alpha'_2$. Recall that $t_1$ does not traverse any switch of $\mathcal{S}$ in $\alpha_1$. Assume that $t_2$ never traverses some switch of $\mathcal{S}$ in $\alpha'_2$. Then, if we let $t$ run solo all the way through the network starting from the final state of $\alpha'_2$, it would encounter the same switches in the same internal states as it would starting from $q_0$. Therefore, $t$ takes the even value $v$. However, one of the tokens $t$, $t_2$ must take an odd value. These imply that $t_2$ traverses at least one switch of $\mathcal{S}$ in $\alpha'_2$. Take $\alpha_2$ to be the shortest prefix of $\alpha'_2$ such that $t_2$ is in front of a switch of $\mathcal{S}$ at the final state $q_2$ of $\alpha_2$.

We now present the details of the formal proof, where we generalize the above arguments for any switching network and any number of processes.

**Proof:** By induction on $i$, $1 \leq i \leq n-1$.

**Basis Case**

Consider the $t_1$-solo, $t_1$-complete execution fragment $\alpha'_1$ in which $t_1$ enters the network in state $q_0$, and runs solo all the way through the network. Clearly, when $t_1$ emerges, the network state $q'_1$ is again quiescent, and $t_1$ takes value $v$.

Assume, by way of contradiction, that $t_1$ never traverses a switch of $\mathcal{S}$ in $\alpha'_1$. Consider now the $t$-solo, $t$-complete execution fragment $\alpha''_1$ in which $t$ enters the network in state $q'_1$, and runs solo all the way through the network. Since $t$ and $t_1$ traverse disjoint sets of switches, $t$ encounters the same switches in the same internal states as it would starting from $q_0$, so when $t$ emerges, the network state is again quiescent, and $t$ takes value $v$ as well.

Because $t$ takes value $v$, and $v \neq g(v)$, it follows that $t$ is serialized first. Since $t_1$ is serialized second, it must take value $f(v)$. Because $t_1$ actually took $v \neq f(v)$, we have a contradiction,

It follows that $t_1$ traverses at least one switch of $\mathcal{S}$ in $\alpha'_1$. Let $\alpha_1$ be the shortest prefix of $\alpha'_1$ such that $t_1$ is in front of a switch $s_1 \in \mathcal{S}$ at the final state $q_1$ of $\alpha_1$.

**Induction Step**

Assume inductively that for some $i$, $1 < i \leq n-1$, the claim holds for all $j$, $1 \leq j < i$; that is, there exists an execution fragment $\alpha_j$ with final state $q_j$ starting from state $q_{j-1}$ such that token $t_j$ is in front of a switch $s_j \in \mathcal{S}$ at state $q_j$.

Consider the $t_i$-solo, $t_i$-complete execution fragment $\alpha'_i$ in which $t_i$ enters the network in state $q_{i-1}$, and runs solo all the way through the network; denote by $q'_i$ the final state of $\alpha'_i$. Suppose that $t_i$ never traverses a switch of $\mathcal{S}$ in $\alpha'_i$.

What value does $t_i$ take at the end of $\alpha'_i$? Recall that $t$ has taken no steps, and that tokens $t_1, \ldots, t_{i-1}$ part-way through the network each apply function $g$. If we run all the $t_j$, $1 \leq j < i$, starting from $q'_i$ until they exit the network, the resulting state is quiescent, so $t_i$ must take $g^k(v)$, for some $k$, $0 \leq k < i$. It follows that $t_i$ must have taken $g^k(v)$ at the end of $\alpha'_i$.

Next consider the $t$-solo, $t$-complete execution $\alpha''_i$ in which $t$ enters the network in state $q'_i$, and runs solo all the way through the network. Since $t$ and $t_i$ traverse disjoint sets of switches, and no $t_j$, $1 \leq j < i$, has traversed any switch in $\mathcal{S}$, $t$ encounters the same switches in the same internal states as it would starting from $q_0$, so when $t$ emerges, it takes value $v$.

At the end of $\alpha''_i$, $t$ has taken $v$ and $t_i$ has taken $g^k(v)$, for some $k$, $0 \leq k < i$. If we run the remaining tokens all the way through the network, the network state is quiescent, and the operations can be serialized. By hypothesis, the $n$-orbit of $v$ under $g$ is aperiodic, so $v$ is not equal to $g^j(v)$, for any $j$, $0 < j < n$. It follows that $t$ cannot be serialized after any of the $t_j$, so $t$ must be serialized first. The remaining tokens must take values of the form $g^j(f(v))$, for $0 \leq j < n$. But, as noted, $t_i$ takes $g^k(v)$, which is not equal to $g^j(f(v))$ for any $j$, $0 \leq j < n$, because the $n$-orbits of $v$ and $f(v)$ under $g$ are disjoint by hypothesis.

It follows that token $t_i$ traverses at least one switch of $\mathcal{S}$ in $\alpha'_i$. Let $\alpha_i$ be the shortest prefix of $\alpha'_i$ such that $t_i$ is in front of a switch $s_i \in \mathcal{S}$ at the final state $q_i$ of $\alpha_i$. ■

We now use Proposition 3.1 to prove our lower bound.

**Theorem 3.2** *Let $\mathcal{R}$ be a read-modify-write network for a set of functions $F$, where $F$ contains two functions $f$ and $g$ such that $f$ displaces $g$. In any sequential execution of $\mathcal{R}$, a token applying function $f$ must traverse at least $\lceil (n-1)/(c-1) \rceil$ switches, where $c$ is the one-shot contention of the network.*

**Proof:** Proposition 3.1 implies that for each $i$, $1 \leq i \leq n-1$, there exists a $t_i$-solo execution fragment $\alpha_i$ with final state $q_i$ starting from state $q_{i-1}$ such that $t_i$ is in front of a switch $s_i \in \mathcal{S}$ at state $q_i$. Let $\alpha = \alpha_1 \cdot \ldots \cdot \alpha_{n-1}$. Since only $n$ tokens, one per process, are involved in $\alpha$, and they are evenly distributed on the input wires, $\alpha$ is a one-shot execution. Notice that all tokens $t_i$, $1 \leq i \leq n-1$ are in front of switches of $\mathcal{S}$ at the final state of $\alpha$, and that each switch in $\mathcal{S}$ is in the same internal state at states $q_0$ and $q_{n-1}$. Thus, in the $t$-solo, $t$-complete execution fragment starting from state $q_{n-1}$ token $t$ traverses all switches of $\mathcal{S}$. Because $\mathcal{R}$ has one-shot contention $c$, no more than $c-1$ other tokens can be in front of any switch of $\mathcal{S}$ in $\alpha$. Thus, $\mathcal{S}$ must contain at least $\lceil \frac{n-1}{c-1} \rceil$ switches. ■

## Applications

Let $S$ be any non-empty set of integer values. An *$S$-adding network* is a read-modify-write network with set of functions $\{\oplus_a \mid a \in S\}$. An *adding network* is an $S$-adding network where $S = \mathbb{Z}$.

**Corollary 3.3** *Let $\mathcal{A}$ be an $S$-adding network. If $S$ contains $a$ and $b$ such that $b$ does not divide $a$, then every token applying $b$ traverses at least $\lceil (n-1)/(c-1) \rceil$ switches of $\mathcal{A}$ in any sequential execution.*

Notice that distinct non-zero $a$ and $b$ divide one another only if $b = -a$. This corollary states that even if the adding network can add only two distinct non-zero numbers, then as long as one isn't the negative of the other, some token has high latency in sequential executions.

If $S = \mathbb{Z}$ then every non-zero value displaces some other value.

**Corollary 3.4** *Let $\mathcal{A}$ be an adding network. Every token adding a non-zero value traverses at least $\lceil (n-1)/(c-1) \rceil$ switches of $\mathcal{A}$ in any sequential execution.*

An *$S$-multiplying network* is a read-modify-write network with set of functions $\{\otimes_a \mid a \in S\}$. A *multiplying network* is an $S$-multiplying network where $S = \mathbb{Z}$.

**Corollary 3.5** *Let $\mathcal{M}$ be an $S$-multiplying network. If $S$ contains $a$ and $b$ such that $a$ is not a power of $b$, then every token multiplying by $b$ traverses at least $\lceil (n-1)/(c-1) \rceil$ switches of $\mathcal{M}$ in any sequential execution.*

If $S = \mathbb{Z}$, then every value distinct from $\pm 1$ or zero displaces some other value.

**Corollary 3.6** *Let $\mathcal{M}$ be a multiplying network. Every token multiplying by a value distinct from $\pm 1$ or zero traverses at least $\lceil (n-1)/(c-1) \rceil$ switches of $\mathcal{M}$ in any sequential execution.*

An *$S$-union network* is a read-modify-write network whose state is a multiset of values from $S$, and whose set of functions is $\{\uplus_a \mid a \in S\}$, where $\uplus_a(U) = U \uplus \{a\}$.

**Corollary 3.7** *Let $\mathcal{U}$ be an $S$-union network. If $S$ contains two distinct elements, then every token traverses at least $\lceil (n-1)/(c-1) \rceil$ switches of $\mathcal{U}$ in any sequential execution.*

An *$S$-concat network* is a read-modify-write network whose state is a sequence of values from $S$, and whose set of functions is $\{\odot_a \mid a \in S\}$, where $\odot_a(\sigma) = \sigma \cdot a$.

**Corollary 3.8** *Let $\mathcal{Q}$ be an $S$-concat network. If $S$ contains two distinct elements, then every token traverses at least $\lceil (n-1)/(c-1) \rceil$ switches of $\mathcal{Q}$ in any sequential execution.*

## 4 Upper Bound

How tight is the lower bound of Section 3? It follows from prior results [10] that any read-modify-write network that can solve consensus for $c$ processes has one-shot contention $c$. This observation implies, for example, that there is no non-trivial network implementation of a `compare&swap` read-modify-write register, because such a construction has an unbounded consensus number [13].

We can, however, show that our bound is essentially tight for an important class of read-modify-write registers. A family of functions $F$ is *commutative* if, for every $f$ and $g$ in $F$, and every value $v$ in the domain $D$, $f(g(v)) = g(f(v))$. Read-modify-write registers with commutative functions include `fetch&inc`, `fetch&add`, and `fetch&mul`. All such read-modify-write registers have consensus number 2 [13], and they are natural candidates for a matching upper bound.

We present a low-contention read-modify-write network, called LADDER, such that in any of its sequential executions tokens traverse $O(n)$ switches; moreover, the average latency of the network is $O(n)$. The switching network described here has the same topology as the SKEW counting network [15], though its behavior is substantially different.

### 4.1 The LADDER Read-Modify-Write Network

A `Ladder` *layer* is an unbounded-depth switching network consisting of a sequence of *binary* switches $s_i$, $i \geq 0$. For switch $s_0$, both input wires are input wires to the layer, while for each switch $s_i$, $i > 0$, the first (or *north*) input wire is an output wire of switch $s_{i-1}$, while the second (or *south*) input wire is an input wire of the layer. The north output wire of any switch $s_i$, $i \geq 0$, is an output wire of the layer, while the south output wire of $s_i$ is the north input wire of switch $s_{i+1}$ (see Figure 2(a)).

A `Ladder` *switching network* of *layer depth* $d$ is a switching network constructed by layering $d$ `Ladder` layers so that the $i$-th output wire of the

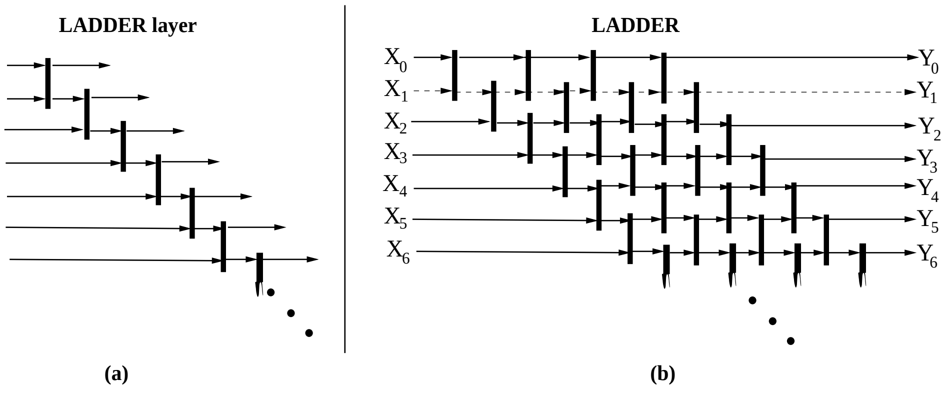

Figure 2: (a) A `Ladder` layer. (b) The `Ladder` switching network of layer depth 4.

one is the $i$-th input wire to the next. Clearly, the `Ladder` switching network has an infinite number of input and output wires. The LADDER *read-modify-write network* consists of a conventional counting network followed by a `Ladder` switching network of layer depth $n$. For the rest of this section, we denote by `Ladder` the `Ladder` switching network; we use LADDER to denote the combined network consisting of `Ladder` and the conventional counting network.

Figure 2(b) illustrates a `Ladder` switching network of layer depth 4. Switches are represented by fat vertical lines, while wires by horizontal arrows. Wires $X_0, X_1, \ldots$, are the input wires of the network, while $Y_0, Y_1, \ldots$, are its output wires. All dashed wires belong to *row* 1.

When writing pseudocode, we assume a reasonably compact encoding for the function defined by any composition of functions in $F$. (For example, in an adding network implementation, we can represent $+_a$ as the integer $a$, the composition of $+_a$ and $+_b$ as $a+b$, and so on.) We use $\circ$ to denote functional composition: $(f \circ g)(v) = f(g(v))$. We assume also that the register has a distinguished initial value $v_0$. For any integer $k > 0$ and for any set of $k$ functions $\{f_1, \ldots, f_k\}$, we denote $\bigcirc_{i=1}^{k} f_i = f_1 \circ f_2 \circ \ldots \circ f_k$.

Each process moves its token through the counting network first, and uses the result to choose an input wire to the `Ladder` network. The counting network ensures that each input wire of `Ladder` is chosen by exactly one token, and each switch is visited by two tokens. A *fresh* switch is one that has never been visited by a token. Each switch $s$ has the following state: a bit *s.toggle* that assumes values `north` and `south`, initially `north`, and a function *s.fn*. The fields *s.north* and *s.south* are immutable pointers to the switches connected to $s$ through its north and south output wires, respectively.

Each token $t$ has the following state: the *t.arg* field is the original function applied by the token. The *t.fn* field is originally the identity function (denoted $\mathbf{1}$). This field accumulates the composition of the functions applied by tokens serialized before $t$. The *t.wire* field records whether the token will enter the next switch on its north or south input wire.

Within `Ladder`, a token proceeds in two *epochs*, a *north* epoch followed by a *south* epoch. Tokens behave differently in different epochs. A token's north epoch starts when the token enters `Ladder`, continues as long as it traverses fresh switches, and ends as soon as it traverses a non-fresh switch. When a north-epoch token visits a fresh switch, the following occurs atomically (1) *s.toggle* flips from `north` to `south`, and (2) *s.fn* is set to the composition of *t.fn* and *t.arg*. Then, $t$ exits on the switch's north wire.

The first time a token $t$ visits a non-fresh switch, it sets $t.fn$ to the composition of that switch's function with *t.fn*, exits on the south wire, and enters its south epoch. Once a token enters its south epoch, it never moves up to a lower-numbered row. When a south-epoch token enters a switch on its south wire, it simply exits on the same wire (and same row), independently of the switch's current state. When a south-epoch token enters a switch on its north wire, it does the following. If the switch is fresh, then, as before, it atomically sets the switch's function to the composition of its function and argument, flips the toggle bit, and exits on the north wire (same row). If the switch is not fresh, it composes the switch's function with its own, and exits on the south wire (one row down). When the token emerges from `Ladder`, the value it returns is constructed by applying its function to the initial value $v_0$.

All tokens other than the one that exits on the first output wire eventually reach a non-fresh switch. When a token $t$ encounters its first non-fresh switch, then that switch's function is the composition of the functions applied by all the tokens that will precede $t$ (so far) in the read-modify-write order. Each time the token enters a non-fresh switch on its north wire, it has been overtaken by an earlier token, so it moves down one row and composes this other token's function with its own.

Very informally, the token knows the set of functions applied by tokens serialized before it. It does not know the order in which these functions were applied. (If it did, it could solve $n$-consensus, which is impossible.) This lack of ordering information is precisely why we need the commutativity requirement.

Figure 3 shows pseudo-code for the two epochs. For ease of presentation, the pseudo-code shows the switch complementing its *toggle* field and updating its *function* field in one atomic operation. However, a slightly more complicated construction can realize this state change as a simple atomic complement operation on the toggle bit.

## 4.2 Correctness

In this section, we prove that `Ladder` is a read-modify-write network. We start with some useful definitions.

The *path* of a token $t$ in an execution $\alpha$, denoted $p_t = s_1 \dots s_m$, $m > 0$, is the sequence of consecutive switches of `Ladder` traversed by $t$ in $\alpha$. For any integer $i$, $1 \leq i < m$, we denote by $\texttt{succ}_t(s_i)$ the switch traversed by $t$ immediately after $s_i$ in $\alpha$. Moreover, for any integer $i$, $1 < i \leq m$, we denote $\texttt{prev}_t(s_i)$ the switch traversed by $t$ immediately before $s_i$ in $\alpha$. For any state $q$ of $\alpha$, we denote by $p_t \uparrow q$, the prefix of $p_t$, containing those switches traversed by $t$ until (but not including) state $q$. We denote by $p_t \downarrow q$ the remaining suffix. We say that the paths of two tokens *cross* if there exists a switch $s$ contained in both paths such that the one token arrives on its north input wire and exits on its south output wire, while the other arrives on its south input wire and exits on its north output wire.

A token that arrives on an input wire of a switch can either *access* the switch (lines 2-3 of `north_traverse`, and lines 7-8, 11 of `south_traverse`) or *ignore* the switch (lines 1-3 of `south_traverse`). A token is *diverted south* on a switch $s$ if it accesses $s$ and exits on the south output wire of $s$. Clearly, a token can be diverted south only if another token has already accessed $s$. Moreover, a token ignores a switch only through its south input wire.

We use the following notation. We denote by $t.fn_q$ the value of variable $t.fn$ of some token $t$ at the beginning of state $q$. We further denote by $s.fn_q$, the value of variable $s.fn$ of a switch $s$ at the beginning of state $q$.

We first outline the major ideas of our proof. We prove that Ladder is a read-modify-write network in five steps:

1. Our first goal is to prove that at most one token traverses any wire of `Ladder` (Proposition 4.2). This proposition is useful in proving all our subsequent statements.

2. In our second step, we prove in Proposition 4.4 that all tokens other than the one that exits on the first output wire, eventually reach a

```
void north_traverse(token $t$, switch $s$) {
(1)     if ($s.toggle$ == NORTH) { /* fresh */
            atomically {
(2)             $s.toggle$ = SOUTH;
(3)             $s.fn = compose(t.fn, t.arg)$;
            }
(4)         north_traverse($t$, $b.north$);
(5)     }   else { /* not-so-fresh */
(6)         $t.fn = compose(t.fn, s.fn)$;
(7)         $t.wire$ = NORTH;
(8)         south_traverse($t$, $b.south$);
        }
}

void south_traverse(token $t$, switch $s$) {
(1)     if ($t.wire$ == SOUTH) { /* ignore switch */
(2)         $t.wire$ = NORTH; /* toggle wire */
(3)         south_traverse($t$, $s.south$);
(4)     } else {
(5)         $t.wire$ = SOUTH; /* toggle wire */
(6)         if ($s.toggle$ == NORTH) { /* fresh */
                atomically {
(7)                 $s.toggle$ = SOUTH;
(8)                 $s.fn = compose(t.fn, t.arg)$;
                }
(9)             south_traverse($t$, $s.north$);
(10)        } else { /* overtaken */
(11)            $t.fn = compose(t.fn, s.fn)$;
(12)            south_traverse($t$, $s.south$);
            }
    }
}
```

Figure 3: Pseudo-Code for `Ladder` Traversal.

non-fresh switch on which they are diverted south. Thus, all such tokens execute their south epoch when exiting the network.

3. We next prove in Proposition 4.7 that each time a token $t$ is diverted south on a switch $s$ reaching some row $k$, it accumulates the composition of the functions of all tokens that exit on lower-numbered output wires than $k$ of the `Ladder` layer in which $s$ resides.

4. Our next goal is to prove (Proposition 4.10) that a similar claim holds whenever a token $t$ exits on a south output wire of a switch by ignoring the switch.

5. In our final step, we consider a token $t$ exiting on output wire $k$ of the network. We use Proposition 4.4 proved in step 2 to argue that $t$ exits on the south output wire of the previous to the last switch it traverses. This switch resides in the last `Ladder` layer and its south output wire is on row $k$; recall that the output wires of this layer are the output wires of the network (see Figure 2(b)). We then employ either Proposition 4.7 proved in step 3 or Proposition 4.10 proved in step 4 (depending on whether $t$ is diverted south or it ignores this switch), to prove that exactly one token exits on each of the lower-numbered output wires than $k$ of the network, and that token $t$ returns the composition of the functions applied by these tokens. Since $t$ exits on the north output wire of the last switch it traverses, these two claims combined imply the read-modify-write property of the network. We further conclude that all tokens exiting on lower-numbered output wires than $k$ cannot enter the network after $t$ has exited it, which implies that LADDER is additionally a linearizable read-modify-write network.

We now present the details of our analysis. In order to prove the claim of our first step, we first prove that `Ladder` is a 1-balancing network.

**Lemma 4.1** *`Ladder` is a 1-balancing switching network.*

**Proof:** Let $s$ be any arbitrary switch of `Ladder`. We prove that if at most one token reaches each input wire of $s$, then at most one token exits on each of its output wires. The only interesting case is when two tokens $t$ and $t'$ reach the two input wires of $s$. Assume without loss of generality that $t$ is the first token traversing $s$.

If $t$ reaches $s$ on its north input wire, it accesses $s$ and exits on its north output wire. Token $t'$, arriving on the south input wire of $s$, either ignores or accesses $s$. In both cases it exits on its south output wire.

Assume now that $t$ reaches $s$ on its south input wire. Token $t$ either ignores or accesses $s$. Either way, token $t'$, arriving on the north input wire, accesses $s$. In case $t$ ignores $s$, it exits on its south output wire, and token $t'$ is the first token to access $s$ and so it exits on the north output wire of $s$. If $t$ accesses $s$, it exits on its north output wire, while token $t'$ exits on its south output wire. ■

Recall that each token first traverses the conventional counting network obtaining a unique value, and then enters on a particular input wire of `Ladder` depending on this value. Therefore, exactly one token enters on each input wire of `Ladder`. Recall that if at most one token enters on each input wire of a 1-balancing network, then at most one token traverses each of its wires. Thus, Lemma 4.1 implies that:

**Proposition 4.2** *For any execution $\alpha$, at most one token traverses each wire of `Ladder`.*

The proof of the first step is complete. In order to prove the claim of our second step, we first prove that if a token enters `Ladder` on input wire $\ell$ and exits it on output wire $k$, all later tokens that enter the network on higher input wires than $\ell$ exit on higher output wires than $k$.

**Lemma 4.3** *Let $t$ be any token that enters `Ladder` on input wire $\ell \geq 0$ and exits on output wire $k \geq 0$ at some state $q_m$. Then, any token $t'$ entering `Ladder` on input wire $\ell' > \ell$ at some later state $q_{m'}$, $m' > m$, exits the network on some output wire $k' > k$.*

**Proof:** Assume, by way of contradiction, that $k' < k$. Then $p_t$ and $p_{t'}$ must cross. Let $s$ be the first switch on which $p_t$ and $p_{t'}$ cross. It must be that $t$ reaches $s$ on its north input wire, since, otherwise, the two paths have crossed already. Recall that $t$ is the first token to traverse $s$. Therefore, $t$ accesses $s$, and exits on its north output wire. Since $t$ reaches $s$ on its north input wire and exits on its north output wire, $p_t$ and $p_{t'}$ cannot cross on $s$. A contradiction. ■

We are now ready to prove that all tokens exited `Ladder` on output wires other than zero have been diverted south at least once during their `Ladder` traversal.

**Proposition 4.4** *Consider a token $t$ which exits* `Ladder` *on output wire $k > 0$ at some state $q_m$, $m > 0$. Then, there exists some earlier state $q_{m'}$, $m' < m$, such that $t$ is diverted south on some switch $s$ at $q_{m'}$.*

**Proof:** Assume, by way of contradiction, that there is no state at which $t$ is diverted south. Let $\ell$ be the input wire on which $t$ enters `Ladder`. If $\ell < n+1$, $t$ goes north through $n$ switches and exits on output wire 0. Recall that $k > 0$. A contradiction.

Consider now the case that $\ell \geq n + 1$. Clearly, $t$ goes north through $n$ switches and exits on output wire $k = l - n$. Since $t$ enters on input wire $\ell$, $t$ obtained value $\ell$ from the conventional counting network. For conventional counting networks ([15, Lemma 2.1]), it holds that at any execution $\alpha$, if a token $t$ exits the network at state $q$ taking a value $\ell$, then for any value $\ell' < \ell$ which has not yet been given, there exists a token traversing the network at $q$ which takes $\ell'$. Therefore, for all values less than $\ell$, either each such value has already been taken by some token or it will be taken by a token currently traversing the conventional counting network. Since there exist $n$ processes in the system, there exists at most $n - 1$ tokens such that either they are still in the conventional network so they will obtain a value less than $\ell$, or they have entered `Ladder` on lower input wires than $\ell$ and have not yet exited. Thus, at least $\ell - n + 1$ tokens have entered and exited `Ladder` before $t$ entered it. Lemma 4.3 implies that all these tokens have exited on lower-numbered output wires than $k$. Proposition 4.2 implies that at most one token has exited on each of the output wires of the network. Therefore, there must be at least one token $t'$ which has exited the network on a higher output wire than $k = l - n$ before $t$ enters the network. Since $t$ enters the network on a higher-numbered input wire than $t'$, Lemma 4.3 implies that $t$ will never reach output wire $k$. A contradiction. ■

We continue to the third step of our analysis. Notice that if a token $t$ is diverted south at some state $q_m$, reaching row $k$, it must either exit or it enters its south epoch at $q_m$, and it remains there for all subsequent states. The only way for token $t$ to move to lower-numbered rows is to reach the south input wire of a switch and exit on its north output wire. However, $t$ ignores all switches that it reaches on their south input wires and, therefore, it never moves to a lower-numbered wire again, as stated by the following lemma.

**Lemma 4.5** *Consider a token $t$ that is diverted south on a switch $s$ at some state $q_m$, $m \geq 0$, reaching row $k$, $k > 0$. Then, for all lower-numbered rows*

*$i$, $0 < i < k$, and for all subsequent states $q_{m'}$, $m' > m$, $t$ is not on row $i$ at state $q_{m'}$.*

Lemma 4.5 implies that if a token $t$ is on row 0 at some state $q_m$, then it has never been diverted south by state $q_m$, since otherwise, it would have never reached row 0 again. Thus, $t$ executes its north epoch at $q_m$. Recall that as long as $t$ is in its north epoch *t.fn* remains unchanged. Therefore:

**Corollary 4.6** *Let $t$ be any token that is on row 0 at some state $q_m$, $m \geq 0$. Then,* (1) *there exists no state $q_{m'}$, $0 \leq m' \leq m$, at which $t$ is diverted south, and* (2) $\mathrm{t.fn}_{q_m} = \mathbf{1}$.

We are now ready to prove that each time a token $t$ is diverted south on a switch $s$ reaching row $k$, it learns (and stores in variable *t.fn*) the composition of the functions of all tokens that exit on lower-numbered output wires than $k$ of the `Ladder` layer in which $s$ resides.

**Proposition 4.7** *Consider a token $t$ which is diverted south on a switch $s$ at some state $q_m$ reaching row $k$, $k \geq 0$, and let $s$ reside in* `Ladder` *layer $\mathcal{L}$. Then,*

(1) *for each integer $i$, $0 \leq i \leq k-1$,*

  a. *there exists exactly one token $t_i$ which exits on output wire $i$ of layer $\mathcal{L}$;*

  b. *either $t_i$ is in* `Ladder` *or it has exited the network by state $q_m$;*

(2) *it holds that $t.fn_{q_{m+1}} = \bigcirc_{i=0}^{k-1} t_i.arg$.*

**Proof:** By induction on $k$, the row number on which $t$ exits south. Clearly, there exists no switch having its south output wire on row 0. Thus, the claim holds vacuously for $k = 0$. Assume inductively that for some fixed integer $k > 0$, the claim holds for all tokens that are diverted south on row $k-1$. We prove that the claim holds for tokens that are diverted south on row $k$.

Let $t$ be any such token and let $s$ be the switch on which $t$ is diverted south at state $q_m$. Since $t$ is diverted south on $s$, it must be that another token $t'$ has accessed $s$ at some earlier state $q_{m'}$, $m' < m$, and has exited on $s$'s north output wire. Clearly, $t' = t_{k-1}$. Since $q_{m'}$ precedes $q_m$, either $t'$ is still in `Ladder` at $q_m$ or it has exited the network by $q_m$. Since $t$ is diverted south on $s$, it updates its function at $q_m$ (that is, it executes either line 6 of `north_traverse` or line 11 of `south_traverse`); therefore, $t.fn_{q_{m+1}} = t.fn_{q_m} \circ s.fn_{q_m}$. We have two cases to consider depending on whether $t$ reaches $s$ on its north or on its south input wire.

1. **Token $t$ reaches $s$ on its north input wire.**

   Clearly, $t'$ reaches $s$ on its south input wire, and it exits on its north output wire. Therefore, $t'$ executes its north epoch. Proposition 4.2 implies that no token other than $t$ subsequently traverses $s$. Therefore, $s.fn_{q_m} = t_{k-1}.arg$ (line 3 of `north_traverse`). We consider two cases depending on whether $k = 1$ or $k > 1$.

   Assume first that $k = 1$. Since $t'$ is the only token to exit on the first output wire of $\mathcal{L}$, and state $q_{m'}$ precedes state $q_m$, condition (1) follows. Since $t$ reaches $s$ on its north input wire, $t$ is on row 0 at state $q_m$. Therefore, Corollary 4.6 implies that $t.fn_{q_m} = \mathbf{1}$. Recall that $t.fn_{q_{m+1}} = t.fn_{q_m} \circ s.fn_{q_m}$, and $s.fn_{q_m} = t_0.arg$. Hence, $t.fn_{q_{m+1}} = t_0.arg = \bigcirc_{i=0}^{k-1} t_i.arg$, as needed by (2).

   Assume next that $k > 1$. Then, $t$ must have exited south on switch $s' = \texttt{prev}_t(s)$ at some state $q_{m_0}$, $m_0 < m$. Clearly, $s'$ is in the same layer as $s$, and $s'$'s south output wire is on row $k - 1$. Thus, by the induction hypothesis, for each $i$, $0 \leq i \leq k - 2$, there exists exactly one token $t_i$ which exits on output wire $i$ of $\mathcal{L}$. Moreover, either $t_i$ is in `Ladder` at $q_{m_0}$ or it has exited the network by $q_{m_0}$. Since $m_0 < m$, it follows that either $t_i$ is in `Ladder` at $q_m$ or it has exited the network by $q_m$. Since $t'$ is the only token to exit on output wire $k - 1$ of $\mathcal{L}$ at $q_{m'}$, and $m' < m$, condition (1) follows. Moreover, by the induction hypothesis,

   $$t.fn_{q_{m_0+1}} = \bigcirc_{i=0}^{k-2} t_i.arg.$$

   Since the next step taken by $t$ is to traverse $s$ at state $q_m$, it follows that

   $$t.fn_{q_m} = \bigcirc_{i=0}^{k-2} t_i.arg.$$

   Recall that

   $$t.fn_{q_{m+1}} = t.fn_{q_m} \circ s.fn_{q_m},$$

   and $s.fn_{q_m} = t_{k-1.arg}$. Therefore,

   $$t.fn_{q_{m+1}} = \bigcirc_{i=0}^{k-2} t_i.arg \circ t_{k-1}.arg = \bigcirc_{i=0}^{k-1} t_i.arg,$$

   as needed by (2).

2. **Token $t$ reaches $s$ on its south input wire.**

   Clearly, $s$ is the first switch on which $t$ is diverted south (otherwise $t$ would ignore $s$). Thus, $t$ executes its north epoch at state $q_m$, so that $t.fn_{q_m} = \mathbf{1}$. Recall that token $t'$ has accessed $s$ at some earlier state

$q_{m'}$. Therefore, $s.fn_{q_{m'+1}} = t'.fn_{q_{m'}} \circ t'.arg$. Since no token traverses $s$ until state $q_m$, it follows that

$$s.fn_{q_m} = t'.fn_{q_{m'}} \circ t'.arg.$$

We again consider two cases depending on whether $k = 1$ or $k > 1$.

Assume first that $k = 1$. Then, $t' = t_0$, and condition (1) follows. Since $t'$ reaches $s$ on its north input wire, $t'$ is on row 0 at state $q_{m'}$, and Corollary 4.6 implies that $t'.fn_{q_{m'}} = \mathbf{1}$. Thus, $s.fn_{q_m} = t_0.arg$. Recall that

$$t.fn_{q_{m+1}} = t.fn_{q_m} \circ s.fn_{q_m}.$$

Hence,

$$t.fn_{q_{m+1}} = t_0.arg = \bigcirc_{i=0}^{k-1} t_i.arg,$$

as needed by (2).

Assume now that $k > 1$. Due to the topology of `Ladder`, $t'$ must have exited south on switch $s' = prev_{t'}(b)$ at some state $q_{m'_0}$, $m'_0 < m' < m$. Clearly, $s'$ is in the same layer as $s$ and $s'$'s south output wire is on row $k-1$, so that by induction hypothesis, for each integer $i$, $0 \leq i \leq k-2$, there exists exactly one token $t_i$ that exits on output wire $i$ of layer $\mathcal{L}$. Moreover, either $t_i$ is in `Ladder` at $q_{m'_0}$ or it has exited the network by $q_{m'_0}$. Since $m'_0 < m$, it follows that either $t_i$ is in `Ladder` at $q_m$ or it has exited the network by $q_m$. Recall that $t'$ is the token exiting on output wire $k-1$ of $\mathcal{L}$, and $m' < m$, so that condition (1) follows.

Moreover, by induction hypothesis, $t'.fn_{q_{m'_0+1}} = \bigcirc_{i=0}^{k-2} t_i.arg$. Since the next step taken by $t' = t_{k-1}$ is to traverse $s$ at state $q_{m'}$, we have

$$t'.fn_{q_{m'}} = t'.fn_{q_{m'_0+1}} = \bigcirc_{i=0}^{k-2} t_i.arg.$$

Recall that $s.fn_{q_m} = t'.fn_{q_{m'}} \circ t'.arg$. Therefore,

$$s.fn_{q_m} = \bigcirc_{i=0}^{k-2} t_i.arg \circ t_{k-1}.arg = \bigcirc_{i=0}^{k-1} t_i.arg.$$

Recall that $t.fn_{q_{m+1}} = t.fn_{q_m} \circ s.fn_{q_m}$, while $t.fn_{q_m} = \mathbf{1}$. Thus,

$$t.fn_{q_{m+1}} = \bigcirc_{i=0}^{k-1} t_i.arg,$$

as needed by (2).

At this point the proof of Proposition 4.7 is complete. ■

Our goal now is to prove that a similar claim holds for the case that a token $t$ exits on the south output wire of a switch $s$ by ignoring $s$. This is done in three steps. We first prove in Lemma 4.8 that every time a token ignores a switch exiting on its south output wire on some row k, the token must have been diverted south at some earlier state on a switch whose output wire is also on row k. Next, we prove in Lemma 4.9 that whenever a token $t$ is diverted south on some switch $s$ of a layer $\mathcal{L}$ reaching some row $k$, the "remaining" path of $t$ and the "remaining" path of any token exiting on a lower than $k$ output wire of $\mathcal{L}$ never cross. Using Proposition 4.7, and Lemmas 4.8 and 4.9, it is then easy to prove the desired claim in Proposition 4.10.

**Lemma 4.8** *Consider any token $t$ which ignores a switch $s$ at some state $q_m$, $m > 0$, and assume that the south output wire of $s$ is on row $k$, $k \geq 0$. Then,*

(1) *there exists some switch $s'$, whose south output wire is on row $k$, on which $t$ is diverted south at some earlier state $q_{m'}$, $0 \leq m' < m$, and*

(2) *for all intermediate states $q_{m''}$, $m' < m'' \leq m+1$, $t$ is on row $k$ at state $q_{m''}$, and $t.fn_{q_{m''}} = t.fn_{q_{m+1}}$.*

**Proof:** We start by proving (1). Since $t$ ignores $s$ at state $q_m$, it is on row $k$ and executes its south epoch at $q_m$. Thus, $t$ has been diverted south at least once by $q_m$. If $t$ has never reached a row other than $k$ by $q_m$, let $q_{m'}$ be the last state before $q_m$ at which $t$ is diverted south, and condition (1) follows.

Assume now that $t$ has reached a row other than $k$ by $q_m$. Let $q_{m_0}$, $m_0 < m$, be the last state preceding $q_m$ such that $t$ is not on row $k$ at $q_{m_0}$. By the topology of `Ladder`, $t$ is either on row $k-1$ or on row $k+1$ at $q_{m_0}$. Assume first that $t$ is on row $k-1$. Let $s'$ be the switch traversed by $t$ at $q_{m_0}$, and let $m' = m_0$. Clearly, the south output wire of $s'$ is on row $k$. Token $t$ reaches $s'$ on its north input wire and exits on its south output wire. Therefore, $t$ is diverted south on $s'$, and condition (1) follows.

Consider finally the case that $t$ is on row $k+1$ at $q_{m_0}$. If $s''$ is the switch traversed by $t$ at $q_{m_0}$, $t$ enters on its south input wire and exits on its north output wire. Therefore, $t$ still executes its north epoch at $q_{m_0}$. Since $t$ executes its south epoch at $q_m$, there must exist a switch $s'$, on which $t$ is diverted south (entering its south epoch) at some state $q_{m'}$, $m_0 < m' < m$. Recall that $t$ resides on row $k$ from state following $q_{m_0}$ to $q_m$, so that the south output wire of $s'$ must be on row $k$, and condition (1) follows.

We continue to prove (2). In all cases above, notice that $t$ is on row $k$ for all states $q_{m''}$, $m' < m'' \leq m$. Since $t$ ignores $s$ at $q_m$, it enters its south input wire and exits on its south output wire, and thus $t$ is still on row $k$ at $q_{m+1}$. Moreover, as long as $t$ is on row $k$, it behaves as follows. If it reaches a switch on its north input wire, it exits on its north output wire, and if it reaches a switch on its south input wire it ignores the switch exiting on its south output wire. In both cases, the function variable of $t$ does not change. Therefore, for all states $q_{m''}$, $m' < m'' \leq m$, $t.\mathit{fn}_{q_{m''}} = t.\mathit{fn}_{q_{m+1}}$, and condition (2) follows. ■

We continue with our second lemma, stating that whenever a token $t$ is diverted south on some switch $s$ of a layer $\mathcal{L}$ reaching some row $k$, the "remaining" path of $t$ and the "remaining" path of any token exiting on a lower-numbered output wire of $\mathcal{L}$ never cross.

**Lemma 4.9** *Consider a token $t$ which is diverted south on a switch $s$ at some state $q_m$, $m \geq 0$, reaching row $k$, $k > 0$. Assume that $s$ resides in layer $\mathcal{L}$, and let $t'$ be any token exited on a lower than $k$ output wire of $\mathcal{L}$. If $q_{m'}$, $m' \geq 0$, is the earliest state at which $t'$ is on the output wire of $\mathcal{L}$, then paths $p_t \downarrow q_m$ and $p_{t'} \downarrow q_{m'}$ never cross.*

**Proof:** Assume by way of contradiction that $p_t \downarrow q_m$ and $p_{t'} \downarrow q_{m'}$ cross. Then, there exists at least one switch traversed by $t$ at some later state than $q_m$ which is also traversed by $t'$ at some later state than $q_{m'}$. Let $s'$ be the first such switch, and denote by $q_{m''}$, $m'' > m$, the state at which $t$ traverses $s'$. Token $t$ reaches $s'$ on its south input wire, since otherwise, the two paths must have crossed on some earlier switch, and thus that switch would be the first (instead of $s$) on which the two paths cross. Since $t$ is diverted south at $q_m$ and $m'' > m$, $t$ executes its south epoch at $q_{m''}$. Therefore, $t$ ignores $s'$ exiting on its south output wire, so that the two paths cannot cross. A contradiction. ■

Consider a token $t$ which exits south on some row $k$ ignoring a switch $s$ residing in layer $\mathcal{L}$ at some state $q_m$. Lemma 4.8 implies that $t$ has been diverted south at some earlier state $q_{m'}$, on a switch $s'$ residing on layer $\mathcal{L}'$ whose output wire is also on row $k$; moreover, $t.\mathit{fn}_{q_{m'+1}} = t.\mathit{fn}_{q_m}$. By Proposition 4.7, there exists a set of $k-1$ tokens such that exactly one such token exit on each of the $k$ first output wires of $\mathcal{L}'$. All these tokens are either in `Ladder` or have exited `Ladder` by $q_{m'}$, and $t.\mathit{fn}_{q_{m'+1}}$ equals the composition of the arguments of these tokens. Lemma 4.9 implies that the "remaining" path of any of these tokens and $p_t \downarrow q_{m'}$ do not cross.

Moreover, Lemma 4.1 implies that at least one token traverses any wire of `Ladder`. Therefore, the set of these tokens is the same as the set of tokens exiting on lower-numbered rows than $k$ of $\mathcal{L}$. We can thus conclude:

**Proposition 4.10** *Consider a token $t$ which exits south on a row $k$, $k \geq 0$, ignoring a switch $s$ at some state $q_m$, $m > 0$, and let $s$ reside in layer $\mathcal{L}$. Then,*

(1) *for each integer $i$, $0 \leq i \leq k-1$,*

   a. *there exists exactly one token $t_i$ which exits on output wire $i$ of layer $\mathcal{L}$;*

   b. *either $t_i$ is in `Ladder` or it has exited `Ladder` by state $q_m$;*

(2) *it holds that $t.fn_{s_{m+1}} = \bigcirc_{i=0}^{k-1} t_i.arg$.*

We are now ready to prove our major proposition. Proposition 4.11 states that if a token exits on an output wire $k$ of the network, exactly one token exits on each of the lower-numbered output wires of the network (condition (1/a); moreover, token $t$ returns the composition of the arguments of these tokens (condition (2)). The two claims combined imply the read-modify-write property of the network. Finally, condition (1/b) states that all tokens exiting on lower-numbered output wires than $k$ cannot enter the network after $t$ has exited it, which implies that `Ladder` is additionally a linearizable read-modify-write network.

**Proposition 4.11** *Consider a token $t$ which exits `Ladder` on its output wire $k$, $k \geq 0$, and let $q_m$, $m > 0$, be the state at which $t$ traverses the last switch in its path. Then,*

(1) *for each integer $i$, $0 \leq i \leq k-1$,*

   a. *there exists exactly one token $t_i$ which exits on output wire $i$ of `Ladder`;*

   b. *either $t_i$ is in `Ladder` or it has exited `Ladder` by state $q_m$;*

(2) *it holds that $t.fn_{q_{m+1}} = \bigcirc_{i=0}^{k-1} t_i.arg$.*

**Proof:** We consider two cases depending on whether $k = 0$ or $k > 0$.

Assume first that $k = 0$, so that $t$ exits on output wire 0. Claim (1) holds vacuously. Since $t$ is on row 0 at state $q_{m+1}$, Corollary 4.6 implies that $t.fn_{q_{m+1}} = \mathbf{1}$, as needed by (2).

Assume now that $k > 0$. Let $s$ be the last switch traversed by $t$ at $q_m$. By the topology of `Ladder`, $t$ exits on the north output wire of $s$. Proposition 4.4 implies that $t$ executes its south epoch at $q_m$, so that $t$ ignores all switches it reaches on their south input wire. Since $t$ exits on the north output wire of $s$, it follows that $t$ reaches $s$ on its north input wire. However, the north input wire of $s$ is the south output wire of $\mathtt{prev}_t(s)$. Moreover, this wire is on row $k$. By the topology of `Ladder`, $\mathtt{prev}_t(s)$ resides in the last `Ladder` layer, whose output wires are the output wires of the network. Depending on whether $t$ is diverted south on $s$ or ignores $s$, Proposition 4.7 or Proposition 4.10, respectively, imply that for each integer $i$, $0 \leq i \leq k-1$, there exists exactly one token $t_i$ which exits on output wire $i$ of the network; moreover, either $t_i$ is still in `Ladder` at $q_m$ or it has exited `Ladder` by $q_m$, as needed by condition (1).

Furthermore,

$$t.\mathit{fn}_{q_m} = \bigcirc_{i=0}^{k-1} t_i.\mathit{arg}.$$

At state $q_m$ token $t$ traverses switch $s$ and exits on its north output wire. Therefore, no other token has ever visited $s$ by state $q_m$, so that $s.\mathit{fn}_{q_m} = \mathbf{1}$. Thus,

$$t.\mathit{fn}_{q_{m+1}} = \bigcirc_{i=0}^{k-1} t_i.\mathit{arg},$$

as needed by (2). ■

Proposition 4.11 (conditions (1/a) and (2)) immediately implies that Ladder is a read-modify-write network.

**Theorem 4.12** Ladder *is a read-modify-write network.*

Proposition 4.11 (condition (1/b)) immediately implies that Ladder is a linearizable read-modify-write network:

**Theorem 4.13** Ladder *is a linearizable read-modify-write network.*

## 4.3 Performance Analysis

In this section we present the performance analysis of Ladder. More specifically, we prove that the average latency of the network is $O(n)$. Moreover, we prove that in any sequential execution of Ladder, all tokens traverse $O(n)$ switches. The performance analysis of Ladder is similar to the one for the Skew network. This follows naturally from the fact that the two networks have the same topology and they both maintain the 1-balancing property. We remark that by knowing just the topology of a switching network, it is not always possible to analyze its performance because the way each token moves in the network may depend on both the state of the token and the state of any switch it traverses.

Fix any (finite or infinite) execution $\alpha$ of LADDER; let $\alpha'$ be any prefix of $\alpha$ involving $k > 0$ tokens. Denote $\alpha''$ the execution that results from $\alpha'$ by letting all tokens still in LADDER at the final state of $\alpha'$ exit the network; clearly, the final state of $\alpha''$ is a quiescent state. Proposition 4.11(1) implies that at the final state of $\alpha''$, all tokens have exited on the $k$ lower output wires. Lemma 4.5 implies that none of the $k$ tokens ever reached any higher-numbered row than $k$. Each of the $k$ rows of `Ladder` traverses $2n$ switches, while the depth of the conventional counting network is less than $n$. Thus, the average latency of $\alpha''$, and therefore also of its prefix $\alpha'$ is $O(n)$. Since $\alpha'$ is any arbitrary prefix of $\alpha$, we can thus conclude:

**Theorem 4.14** *The average latency of* LADDER *is* $O(n)$.

We finally prove that in any sequential execution of LADDER, all tokens traverse $O(n)$ switches. We first prove the following proposition.

**Proposition 4.15** *In any sequential execution $\alpha$ of* `Ladder`*, token $t_k$ that exits the network on output wire $k$, $k \geq 0$, enters on input wire $k$ and it never moves on any row other than $k$ during its traversal.*

**Proof:** By induction on $k$. For the basis case, let $t_0$ be the first token traversing the network; $t_0$ enters on input wire 0 and no other token enters the network before $t_0$ exits it. Thus, $t_0$ moves always north and never moves on a row other than 0, as needed. Assume, inductively, that the claim holds for some fixed integer $k \geq 0$. We prove that the claim holds for token $t_{k+1}$ that exits the network on output wire $k+1$. Since $\alpha$ is a sequential execution, $t_{k+1}$ enters on input wire $k+1$ of `Ladder`. By induction hypothesis and by the topology of `Ladder` all switches reached by $t_{k+1}$ through their south input wire have been traversed by token $t_k$. Therefore, $t_{k+1}$ exits on their south output wire. Moreover, all switches reached by $t_{k+1}$ through their north input wire have not yet been traversed by any other token. Thus, $t_{k+1}$ exits on their north output wire. We conclude that $t_{k+1}$ never reaches some row other than $k+1$, as needed.

■

Recall that each row of `Ladder` traverses $2n$ switches. Thus, Proposition 4.15 implies that each token traverses $2n$ switches during its `Ladder` traversal. Since the depth of the conventional counting network is less than $n$, we thus conclude:

**Theorem 4.16** *In any sequential execution of* LADDER*, all tokens traverse $O(n)$ switches.*

By Proposition 4.2, at most one token traverses each wire of `Ladder` in any execution. Since the fan-in of any of its switches is two, it follows that the worst-case contention of `Ladder`, and therefore also its one-shot contention, is $O(1)$. Thus, both the worst-case contention and the one-shot contention of LADDER is bounded by the worst-case contention and the one-shot contention, respectively, of the employed traditional counting network. Lemma 2.2 implies that the one-shot contention of any counting network with $w = n$ and constant $\phi$ is constant. If we choose such a counting network (see e.g., [3]) as the traditional counting network of LADDER, it follows that the one-shot contention of LADDER is constant[3].

## 5 Discussion

LADDER has the same topology as SKEW which is linearizable. We remark that most read-modify-write networks whose set of functions satisfies the displacement property have executions that must be linearizable; however, this is not the case for all their executions[4]. On the other hand, given a linearizable counting network, it is not clear whether one can derive a corresponding switching network supporting commutative or any other kind of interesting read-modify-write operations (beyond fetch-and-inc and fetch-and-dec). It is an interesting open problem to investigate whether any linearizable counting network has a corresponding switching network of the same topology supporting some kind of interesting read-modify-write operations, and vice versa.

Even though `Ladder` has an unbounded number of switches, it can be implemented as a finite network by "folding" the network so that each folded switch simulates an unbounded number of primitive switches. A similar folding construction appears in [15] for SKEW. The state of any switch of folded LADDER is infinite, because an infinite number of *function* variables

[3]Aspnes *et al.* [3] have proved that the worst-case contention of any finite counting network is $\Theta(n)$. Thus, the worst-case contention of LADDER is $\Theta(n)$ as well.

[4]As an example, an execution of an adding network in which tokens add to the register different powers of two must be linearizable, while an execution of the same network in which all tokens add to the register the same value is not necessarily linearizable.

should be simulated by each switch, but only a finite number of them are "active" at any time.

LADDER is not wait-free and supports only commutative read-modify-write operations. However, our lower bound proves that LADDER, as well as all switching networks supporting a large class of read-modify-write operations have limited practical appeal. Nevertheless, there is some theoretical interest to know whether there exist lock-free and low-contention read-modify-write networks for other interesting classes of functions, and wait-free networks for any interesting class.

To conclude, counting networks exhibit three desirable characteristics: they are lock-free, they have low contention, and low latency. They are therefore considered attractive distributed data structures for implementing the restricted set of `fetch&inc` and `fetch&dec` read-modify-write operations. It was natural to expect that appropriate generalizations of counting networks would be good candidates for equally attractive distributed data structures for other read-modify-write operations. This work proves that this is not the case. It may yet be possible to construct a lock-free, low-latency, and low-contention distributed data structure for these operations, but it will require a radically new approach.

**Acknowledgment:** We would like to thank Faith Fich for many useful comments that improved the presentation of the paper. Our thanks go to the anonymous DISC'01 reviewers for their feedback.

## 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," *Proceedings of the 16th International Symposium on Theoretical Aspects of Computer Science,* pp. 393–403, 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.

[4] H. Britt, S. Moran and G. Taubenfeld, "Public Data Structures: Counters as a Special Case," *Proceedings of the 3rd Israel Symposium on Theory of Computing and Systems,* pp. 98–110, January 1995.

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

[6] C. Busch and M. Herlihy, “A Survey on Counting Networks,” *Proceedings of the Workshop on Distributed Data and Structures,* Orlando, Florida, March 1998.

[7] C. Busch and M. Herlihy, “Sorting and Counting Networks of Small Depth and Arbitrary Width,” *Proceedings of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures,* pp. 64–73, June 1999.

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

[9] 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.

[10] 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.

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

[12] 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.

[13] M. Herlihy, “Wait-Free Synchronization,” *ACM Transactions on Programming Languages and Systems,* Vol. 13, No. 1, pp. 124–149, January 1991.

[14] M. Herlihy, B. Lim and N. Shavit, “Scalable Concurrent Counting,” *ACM Transactions on Computer Systems,* Vol. 13, No. 4, pp. 343–364, November 1995.

[15] M. Herlihy, N. Shavit and O. Waarts, “Linearizable Counting Networks,” *Distributed Computing,* Vol. 9, No. 4, pp. 193-203, 1996.

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

[17] Herlihy M., Wing J., ”Linearizability: a Correctness Condition for Concurrent Objects”. *ACM Trans. on Prog. Lang. and Syst.* 12,3:463-492, 1990.

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

[19] C. 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.

[20] 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 (special issue on Distributed Algorithms).

[21] N. Lynch and M. Tuttle, “Hierarchical Correctness Proofs for Distributed Algorithms,” *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing,* pp. 137–151, August 1987.

[22] 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.

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

[24] N. Shavit and D. Touitou, “Elimination trees and the Construction of Pools and Stacks,” *Theory of Computing Systems,* Vol. 30, No. 6, pp. 645–670, November/December 1997.

[25] Nir Shavit, Asaph Zemach, “Combining Funnels: A Dynamic Approach to Software Combining.” *Journal of Parallel and Distributed Computing* 60(11): 1355-1387 (2000)

[26] N. Shavit and A. Zemach, “Diffracting Trees,” *ACM Transactions on Computer Systems,* Vol. 14, No. 4, pp. 385–428, November 1996.

[27] R. Wattenhofer, "Distributed Computing - How to Bypass Bottlenecks," Ph.D. Thesis, ETH Zurich, 1998.

[28] R. Wattenhofer and P. Widmayer, "An Inherent Bottleneck in Distributed Counting," *Journal of Parallel and Distributed Computing,* Vol. 49, No. 1, pp. 135–145, February 1998.

[29] P. Yew, N. Tzeng and D. Lawrie, "Distributed Hot-Spot Addressing in Large-Scale Multiprocessors," *IEEE Transactions on Computers,* pp. 388-395, April 1987.