THE DISTRIBUTED COMPUTING COLUMN

BY

MARIO MAVRONICOLAS

Department of Computer Science, University of Cyprus
75 Kallipoleos St., CY-1678 Nicosia, Cyprus
mavronic@cs.ucy.ac.cy

AN APPLICATION OF THE MONOTONE LINEARIZABILITY LEMMA*

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

Abstract

Monotone RMW operations are associated with *monotone groups*, a certain class of algebraic groups; Fetch&Add and Fetch&Multiply operations are two popular examples. Recently, Busch *et al.* [3] introduced the *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. Through the *Monotone Linearizability Lemma*, Busch *et al.* [3] derived the *first* lower bounds on *size* (the number of *switches*) for any (non-trivial) *switching network* implementing a monotone RMW operation.

In this note, we present another application of the *Monotone Linearizability Lemma*. We provide a very simple and succinct proof that any switching network implementing a monotone RMW operation has *sequential* executions with n processes and latency $\Omega(n)$. Since Fetch&Increment is implementable with counting networks of polylogarithmic latency [2], this lower bound implies a time complexity separation between Fetch&Increment and any monotone RMW operation in the model of switching networks.

*Work partially supported by the EU contracts IST-1999-14186 (ALCOM-FT) and IST-2001-33116 (FLAGS). †Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY 12180; Email: buschc@cs.rpi.edu. ‡Department of Computer Science, University of Cyprus, Nicosia CY-1678, Cyprus; 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.

1 Introduction

A Read&Modify&Write *shared variable* or *register* [8, 11], henceforth abbreviated as RMW, is an abstract variable type that allows reading its old value, computing via some specific *operator* a new value as a function of the old value, and writing the new value back to the register, all in a single, *atomic* (indivisible) RMW operation. The Fetch&Increment, Fetch&Add and Fetch&Multiply are popular examples of RMW registers.

Due to their fundamental importance as synchronization primitives, it is desirable to devise suitable *distributed data structures* for the construction of high-concurrency, low-latency and 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., [4]). A *counting network* [2] 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 [1].

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.

Busch *et al.* [3] have shown the *Monotone Linearizability Lemma,* which establishes inherent ordering constraints of *linearizability* [10] for a certain class of executions of any distributed system that implements a monotone RMW operation; recall that an execution is *linearizable* [10] if the values returned to operations in it respect the real-time ordering of the operations.

Busch *et al.* [3] used *switching networks* [6] as a test-bed for the applicability of their *Monotone Linearizability Lemma.* Switching networks are a class of distributed data strucrures, recently described by Fatourou and Herlihy [6], that may be used for concurrent, low-contention implementations of RMW registers; switching networks are a natural generalization of *counting networks* [2]. Through a modular use of the *Monotone Linearizability Lemma,* Busch *et al.* [3, Section 6] have shown the the *first* lower bounds on size (the number of *switches* in the network) for any (non-trivial) highly concurrent, low-contention switching network that implements a monotone RMW operation.

In this note, we present yet another application of the *Monotone Linearizabil-*

ity Lemma. This application deals with *latency*, the maximum number of switches traversed by a token in a switching network. We prove that 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 $\left\lceil \frac{n-1}{c-1} \right\rceil$; here, 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.

Counting networks provide concurrent implementations of Fetch&Increment operation achieving *finite* size and latency *polylogarithmic* in the number n of concurrent processes [2]. In contrast, the shown lower bound on latency 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 absence of concurrency). So, it implies a time complexity separation between Fetch&Increment and any monotone RMW operation in the model of switching networks.

Our lower bound on latency complements, via a shorter and simpler proof, a correponding lower bound of $\left\lceil \frac{n-1}{c-1} \right\rceil$ on latency shown in [7, Theorem 3.2].

2 Monotone Groups

This section closely follows [3, Section 2]. Denote Z, $\mathbb{N}$ and Q the sets of integers, natural numbers (including zero), and rational numbers (excluding zero), respectively.

We start with the definition of a group. A (binary) *operator* on a set Γ is a mapping $\oplus : \Gamma \times \Gamma \to \Gamma$. 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 $a \in \Gamma$, $a \oplus e = e \oplus a = a$.

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 $a, b \in \Gamma$, $a \oplus b = b \oplus a$.

We proceed to define two composite operators by applying the (binary) operator $\oplus$ a number of times. For any integer k, define the unary operator $\bigoplus_k : \mathbf{\Gamma} \to \mathbf{\Gamma}$ 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 integer n, the operator $\biguplus_n : \mathbf{\Gamma}^n \to \mathbf{\Gamma}$ is n-ary.

- For $n = 0$, it assumes the constant value $\biguplus_0 = e$.
- For $n = 1$, $\biguplus_1\{a\} = a$ for all elements $a \in \mathbf{\Gamma}$. For $n = -1$, $\biguplus_{-1}\{a\} = a^{-1}$.
- For $|n| \geq 2$. $\biguplus_n$ takes as input an ordered multiset $\{a_1, a_2, \ldots, a_{|n|}\} \in \mathbf{\Gamma}$, and it yields the result

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

  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*.

Assume now that the set $\mathbf{\Gamma}$ is totally ordered; thus, a *total order* $\preceq$ is defined on $\mathbf{\Gamma}$. For any pair of elements $a, b \in \mathbf{\Gamma}$, write $a \prec b$ (and, equivalently, $b \succ a$) if $a \preceq b$ and $a \neq b$. A *monotone subdomain* of $\mathbf{\Gamma}$ is a subset $\mathbf{M} \subseteq \mathbf{\Gamma}$ that satisfies the following three properties:

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

A *monotone group* [3] is a quadruple $\langle \mathbf{\Gamma}, \mathbf{M}, \oplus, \preceq \rangle$, where $\langle \mathbf{\Gamma}, \oplus \rangle$ is an Abelian group, $\preceq$ is a total order on $\mathbf{\Gamma}$, and $\mathbf{M} \subseteq \mathbf{\Gamma}$ is a monotone subdomain of $\mathbf{\Gamma}$. It is easy to see that both quadruples $\langle \mathbf{Z}, \mathbf{N} \setminus \{0\}, +, \leq \rangle$ and $\langle \mathbf{Q}, \mathbf{N} \setminus \{0, 1\}, \cdot, \leq \rangle$ are monotone groups.

Fix any integer $n \geq 2$. Consider n distinct elements $a_1, a_2, \ldots, a_n \in \mathbf{\Gamma}$ with $a_1, a_2, \ldots, a_n \neq e$. Say that $a_1, a_2, \ldots, a_n$ are *n-wise independent over* $\langle \mathbf{\Gamma}, \oplus \rangle$ 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, $\biguplus_{i=1}^{n} \bigoplus_{k_i} a_i \neq e$. Say that the monotone group $\langle \mathbf{\Gamma}, \mathbf{M}, \oplus, \leq \rangle$ is *n-wise independent* [3] if there are n distinct elements $a_1, a_2, \ldots, a_n \in \mathbf{M}$, with $a_1, a_2, \ldots, a_n \neq e$, that are n-wise independent over $\langle \mathbf{\Gamma}, \oplus \rangle$. Busch *et al.* [3, Lemma 2.5] prove that for any integer $n \geq 2$, the monotone group $\langle \mathbf{\Gamma}, \mathbf{M}, \oplus, \leq \rangle$ is n-wise independent.

3 System Model

Our model of a distributed system is the one in [3, Section 3]. We consider a distributed system $\mathbf{P}$ consisting of sequential *processes*. Each process applies a sequence of operations to a distributed data structure, 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 $a \in \mathbf{M}$; each *response* at process p_i has the form $\texttt{Response}_i(b)$ for some value $b \in \mathbf{M} \cup \{e\}$. Formally, an *execution* of system $\mathbf{P}$ is a (possibly infinite) sequence α of *invocation* and *response* events. 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.

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

For any execution α of system $\mathbf{P}$, a *serialization* $S(\alpha)$ [5] of execution α is a sequence whose elements are the operations of α, and each operation of α appears exactly once in $S(\alpha)$. Thus, a serialization $S(\alpha)$ is a total order $\xrightarrow{S(\alpha)}$ on the set of operations in α. Notice that there may be, in general, many possible serializations of the execution α.

Say that a serialization $S(\alpha)$ is *valid for the monotone group* $\langle \mathbf{\Gamma}, \mathbf{M}, \oplus, \leq \rangle$ 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 two of operations $op_{i_1}^{(1)} = [\texttt{Invoke}_{i_1}(a_1),$

$\texttt{Response}_{i_1}(b_1)]$ and $op_{i_2}^{(2)} = [\texttt{Invoke}_{i_2}(a_2), \texttt{Response}_{i_2}(b_2)]$ that are consecutive in $S(\alpha)$, $b_2 = b_1 \oplus a_1$.

System **P** *implements the monotone group* $\langle \mathbf{\Gamma}, \mathbf{M}, \oplus, \preceq \rangle$ if every execution α of **P** has a serialization that is valid for the monotone group. *Monotone* RMW operations are those associated with monotone groups. Say that system **P** implements the (monotone) operation RMW ($\langle \mathbf{\Gamma}, \mathbf{M}, \oplus, \preceq \rangle$) whenever it implements the associated monotone group. In the rest of this section, we refer to a distributed system **P** implementing a monotone group $\langle \mathbf{\Gamma}, \mathbf{M}, \oplus, \preceq \rangle$.

Busch *et al.* [3, Lemma 3.1] prove that for any execution α of system **P**, there is a *unique* serialization $S(\alpha)$ that is valid (for the monotone group). Fix any arbitrary execution α of the system **P**, and its (unique) valid serialization $S(\alpha)$.

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

We continue to state the *Monotone Linearizability Lemma* [3, Proposition 5.1], which establishes ordering constraints of linearizability on the system **P**. Recall that the monotone group $\langle \mathbf{\Gamma}, \mathbf{M}, \oplus, \preceq \rangle$ is n-wise independent for any integer $n \geq 2$. So, there are n distinct elements $a_1, a_2, \ldots, a_n \in \mathbf{M}$, with $a_1, a_2, \ldots, a_n \neq e$, which are n-wise independent over $\langle \mathbf{\Gamma}, \oplus \rangle$.

Proposition 3.1 (Monotone Linearizability Lemma). *Consider any execution* α *of system* **P** *in which each process* p_i *issues only operations of type* a_i*, where* $1 \leq i \leq n$*. Then,* α *is linearizable.*

Although linearizability has so far been studied as a *required* property for a distributed system that best guarantees acceptable concurrent behavior, the *Monotone Linearizability Lemma* provides the *first* (non-trivial) instance of a distributed system where linearizability is an *inherent* property.

4 Switching Networks

In this section, we follow [3, Section 4] to present a framework for switching networks.

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. In either case, a switch changes its internal state according to its *transition function*.

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 $\mathcal{N}$. A *path* in a switching network is a sequence of switches each (other than the last) connected to the next.

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

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

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

In more detail, when a token arrives on an input wire of a switch, the following events occur in a single, *atomic* (indivisible) step: *1*. The switch removes the token from the input wire. *2*. The switch changes state. *3*. The token changes state. *4*. The token is routed to an output wire of the switch.

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*. *Live-*

ness 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. 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* [2].

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.

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

An *execution* of a switching network is a finite or infinite sequence $\alpha = Q_0, \tau_1, Q_1, \tau_2, Q_2, \ldots$ of alternating configurations and switch transitions such that: *1.* Q_0 is the *initial* configuration, in which there are no tokens on input wires of switches except for at least one token on input wires of the network, and all switches are in their initial internal states. 2. For each triple $\langle Q_i, \tau_{i+1}, Q_{i+1} \rangle$, where $i \geq 0$, the switch transition τ_{i+1} carries the configuration Q_i to the configuration Q_{i+1}.

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.

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.

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 net-

work. 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. The *capacity* c [9] 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 switching network $\mathcal{N}$ can be used to implement a monotone group $\langle \mathbf{\Gamma}, \mathbf{M}, \oplus, \preceq \rangle$ as follows. *1.* Each token t by process p_i corresponds to an operation

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

invoked by process p_i, where $a \in \mathbf{M}$ and $b \in \mathbf{M} \cup \{e\}$. 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. *2.* 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. *3.* 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 $\mathcal{N}$. The *switching network $\mathcal{N}$ implements the monotone group* $\langle \mathbf{\Gamma}, \mathbf{M}, \oplus, \preceq \rangle$ if the induced distributed system $\mathbf{P}(\mathcal{N})$ implements the monotone group $\langle \mathbf{\Gamma}, \mathbf{M}, \oplus, \preceq \rangle$.

5 The Impossibility of Low-Latency Switching Networks

In this section, we present an impossibility result for low-latency switching networks that implement monotone groups. We first prove an interesting property of such networks.

Lemma 5.1 (Intersection Lemma). *Consider a switching network $\mathcal{N}$ implementing 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 Q. 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 Q, 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 Q do not intersect. Since the network $\mathcal{N}$ implements the monotone group $\langle \mathbf{\Gamma}, \mathbf{M}, \oplus, \preceq \rangle$, 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 both equal to $v \oplus a$ (those that would be returned in separate executions where only one of the tokens would be running). However, the *Monotone Linearizability Lemma* (Proposition 3.1) implies that execution α', where both tokens are running, is linearizable. 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 $\langle \mathbf{\Gamma}, \mathbf{M}, \oplus, \preceq \rangle$, 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. □

We are now ready to prove:

Theorem 1 (Lower Bound on Latency of Switching Networks). *Take a switching network $\mathcal{N}$ that implements a monotone group $\langle \mathbf{\Gamma}, \mathbf{M}, \oplus, \preceq \rangle$. 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 $a_1, a_2, \ldots, a_n \in \mathbf{M}$ which are n-wise independent in $\langle \mathbf{M}, \oplus \rangle$.

By Lemma 5.1, 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. □

Together with the lower bounds on size shown in [3], Theorem 1 suggest 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.

References

[1] 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).

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

[3] C. Busch, M. Mavronicolas and P. Spirakis, "The Cost of Concurrent, Low-Contention Read&Modify&Write," *Theoretical Computer Science*, accepted.

[4] 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.

[5] 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.

[6] 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.

[7] P. Fatourou and M. Herlihy, "Read-Modify-Write Natworks," *Distributed Computing*, Vol. 17, pp. 33–46, 2004.

[8] 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.

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

[10] 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.

[11] 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.