Concurrent Counting[*]

Shlomo Moran[†] Gadi Taubenfeld[‡] Irit Yadin[§]

June 20 1995

Abstract

In this paper we study implementations of concurrent counters, which count modulo some (large) integer m, using only small valued objects. A concurrent counter is a counter that can be incremented and read, possibly at the same time by many processes. The counters we study do not depend on the initial state of the memory and hence are more robust to memory changes. Also, we assume that all the processes are identical which makes them easy to program. Finally, all the algorithms are required to be wait-free – a correct processor cannot be prevented from finishing its increment or read operations – and thus the algorithms can tolerate any number of process failures. We concentrate on providing upper and lower bounds on the space complexity of the counters studied.

1 Introduction

1.1 The Concurrent Counter Problem

Counters are basic objects which are used in various computer applications. A counter (mod m) holds an integer from $\{0, .., m-1\}$, and has two basic operations: `increment` – which increments the value by one (mod m), and `look` – which gets the current value. A *concurrent* counter is a counter in a shared memory environment, which can be *incremented* and *looked* at, possibly at the same time, by many processes. Throughout the paper we assume that a counter is always incremented modulo m for some fixed (but arbitrarily large) m.

Since counters are used as basic building blocks in many applications, results concerning concurrent counters can be used in solving various problems in asynchronous computation. The implementation of concurrent counters raises many basic problems concerning the possibility and the cost of multi-process coordination in an asynchronous shared memory system. In this paper we consider three types of concurrent counters.

[*]A preliminary version of this work appeared in the *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing*, Vancouver, Canada, August 1992.

[†]Computer Science Department, Technion, Haifa 32000, Israel. Part of this work was done while visiting at AT&T Bell Laboratories, Murray Hill.

[‡]AT&T Bell Laboratories and the Open University of Israel.

[§]Computer Science Department, Technion, Haifa 32000, Israel.

A *static counter* guarantees that a `look` operation returns the correct value when it is not concurrent with any `increment` operation. In the case that a `look` operation overlaps an `increment` operation, a static counter may return an arbitrary value.

A *dynamic counter* guarantees that the counter will hold the correct value even if it is incremented concurrently by several processes and that processes can read a correct value of the counter even if the read is concurrent with other increments or reads. That is, for a given `look` operation, let c_1 be the initial value of the counter plus the number of `increment` operations that were completed before the look operation started, and let c_2 be the initial value of the counter plus the total number of `increment` operations that were initiated before the look operation was completed. Then the `look` operation should return some value between c_1 and c_2.

A *linearizable counter* is a dynamic counter in which the executions of the `increment` and `look` operations are linearizable. That is, it behaves as if each of these operations is atomic ([HW90]). (We will also call these counters atomic counters.)

The most common example of a concurrent counter is probably a global clock, which can be incremented by one process, but arbitrarily many processes may gets its value [Lam90]. In implementing a global clock one would not like to set a bound on the number of processes that may see the time in this clock, though in general only one process is allowed to change its value.

Another example of a possible use of a concurrent counter is in protocols for the wakeup problem [FMRT90]. In some of the protocols for this problem every process starts its participation in the protocol by incrementing a counter. The counters used in the wakeup protocols differ from the one used for global clocks in two important features: first, it can be incremented by many distinct processes; second, in each run of the protocol, the counter of the wakeup protocols can be incremented at most some bounded and known number of times. Other examples are in fault-tolerant solutions for the consensus or leader election problems [Fis83, KMZ84, Pet82]. In such solutions it is sometimes required to count the number of processes that already voted. For example, in [FMT93] a consensus is reached after some process observes that a counter is incremented by more than half of the processes.

It is easy to implement a concurrent counter (mod m) using one register which holds m distinct values, $\{0, ..., m-1\}$. To increment or read the counter a process first *locks* the access to the counter, modifies or gets its value, and then *unlocks* the access to the counter. However, in a system which supports atomic operations only on small shared objects, the implementation become much less trivial. In this paper we study implementations in a system which only supports atomic operations on bits. Our goal is to design solutions to the concurrent counter problem that require only weak atomicity and thus are easy to implement; do not depend on the initial state of the memory and hence are more robust to memory changes; and are wait-free – a correct process can always finish its `increment` or `look` operations regardless of the behavior of the other processes – and thus can tolerate any number of process failures. Finally, and most important, we want our solutions to use as little shared space as possible.

1.2 Computational Model

The model consists of a fully asynchronous collection of identical anonymous deterministic processes that communicate via bounded size shared registers which are initially in an arbitrary unknown state. The registers that are used are *binary* registers unless otherwise indicated. Also, unless otherwise stated, it is assumed that access to a shared register is via atomic "read-modify-write" instructions, which, in a single indivisible step, reads the value of the register and then writes a new value that can depend on the value just read.

In some cases we develop protocols under the assumption that only one process can increment the counter, and in such cases we assume only read/write atomicity. That is, in one atomic instruction a process may either read from or write to a register. A standard argument shows that when two or more identical processes may increment the counter, the read-modify-write atomicity cannot be replaced by the weaker read/write atomicity (intuitively, when using only reads and writes, we can always "hide" the steps of all but one of the identical processes, by running the processes in lock steps).

We assume that any number of processes can fail. The only kind of failures we consider are crash failures, in which a process may become faulty at any time during its execution, and when it fails, it simply stops participating in the protocol.

Assuming an arbitrary unknown initial state relates to the notion of self-stabilizing systems defined by Dijkstra [Dij74]. However, Dijkstra considers only non-terminating control problems such as the mutual exclusion problem, whereas our implementations of counters can also be used to solve decision problems such as the wakeup and consensus problems, in which a process makes an irrevocable decision after a finite number of steps.

It seems that, in some respects, this model more accurately reflects reality, where in many cases all processes are programmed alike, there is no global synchronization, and it is not possible to simultaneously reset all parts of the system to a known initial state. Our model is similar to the shared memory model studied in [FMRT90, FMT93], except that there it is assumed that there is a *single* finite sized shared register, which is also accessed via atomic read-modify-write instructions.

1.3 Summary of Results

We present lower and upper bounds on the number of shared bits required to implement static and dynamic counters as a function of the number of processes that are allowed to increment the counter. In some cases we present implementations that are very efficient in both space and time, while in other cases we show that any implementation must be very inefficient. All the results are stated under the assumption that the basic atomic operations are performed on single binary registers (bits). These results can be improved (in terms of the space used) when larger registers are available. In all our upper bounds we assume that m is a power of 2. As is shown in a more recent paper, this assumption is necessary [MT93]. Finally, the notion of a static (dynamic) n-counter protocol means that no more than n processes are allowed to increment the counter. We may let n be infinity (∞) which means that there is no restriction on the number of processes that can increment the counter. We will also use the notion of an ℓ-*bounded* counter protocol which is a protocol where a total of at most ℓ increments are performed in any of its runs. We give a brief overview of our results below.

Optimal Static ∞-Counters: We present a static ∞-counter protocol that uses only $\log m$ bits. That is, in the protocol there is no bound on the number of processes that are allowed to increment the counter and it matches the trivial $\log m$ bits lower bound.

Optimal Dynamic 1-Counter: In the case when only one process is allowed to increment the counter, we are able to construct a *dynamic* counter protocol that uses only $\log m$ bits, and hence matches the trivial lower bound. Thus, our protocol gives an optimal solution to Lamport's global clock problem. In designing the protocol we use, in a new way, the reflected binary Gray code. Consequently, incrementing the counter requires a single write operation, and hence, unlike other implementations studied in the literature, it consists of a single atomic step in any model that supports read/write atomicity on single bits.

Dynamic Counters for Many Processes: We present an m-bounded dynamic ∞-counter protocol which uses exactly m bits. Then we use this protocol to construct an (unbounded) dynamic n-counter which uses $\alpha(\log m+1)$ shared bits, where α is the smallest power of 2 that is not smaller than n.

Lower bound: Let $k = \min(\frac{m+1}{2}, \frac{n+1}{2}, \sqrt{\frac{\ell+1}{3}})$. We prove that any ℓ-bounded dynamic n-counter protocol must use at least k registers. This result holds even when the processes have unique identifiers and there is only one possible initial state. Furthermore, by making various restrictions on the way processes may increment the counter we are able to tighten these bounds.

1.4 Related Work

In [Lam90], Lamport developed algorithms to implement both monotonic and cyclic multiple-word clocks that are updated by one process and read by one or more processes. Lamport's cyclic clock problem is a special case of the concurrent (dynamic) counter problem where there is only one process that can increment the counter. Lamport's solution uses $2\log m+2$ registers, and relies on the assumption of a known initial value.

Following the (conference) publication of our paper [MTY92], concurrent counting has been further investigated. In [MT93] it is shown that in a model which supports only read-modify-write of single bits, a static counter (modulo m) exists only if $m = 2^k$, where k is bounded from above by the number of bits a process may change during a single increment operations. In [BMT95] two results are proved for a model in which nothing can be assumed in advance about the number or the identities of the processes. The first result fully characterizes the static counters which can be implemented from a given set of atomic counters. The second result shows that it is impossible to construct large dynamic counters from smaller atomic counters. The correctness of the second result in [BMT95] depends on the assumption that the dynamic counter is required to work regardless of the initial values of the atomic counters it is constructed from. In the case where the initial values are known, it is shown in [BIS95] that it is possible to implement a dynamic counter which counts modulo some power of two, using atomic counter which counts modulo two (without assuming anything about the number or the identities of the processes).

Aspnes, Herlihy and Shavit [AHS91] introduced a new class of networks, called *counting networks*. They used counting networks to construct various objects such as a *shared counter* which is an object that can issue the numbers 0 to $m-1$ in response to m requests by processes. Counting networks can be viewed as objects which support one atomic operation, which consists of both `increment` and `look` . It seems that counting networks cannot support the `look` operation of dynamic counters (without incrementing the counter). The two constructions of counting networks in [AHS91] require $O(m \log^2 m)$ binary registers. Our problem seems to be related to but different from counting networks: (1) As mentioned above, it is not clear whether the counting networks presented in [AHS91] can support a `look` operation without incrementing the counter, while we implement a `look` operation in our dynamic counters. (2) All implementations of counting networks rely on the assumption of a single initial state, that is, all shared registers require initialization, while we require that a solution to the concurrent counter problem will work for any possible initial state. Counting networks have been further investigated in [AA92, AVY94, BM94b, HBS92, HSW91, KP92, SZ94].

Concurrent counters and counting networks are data structures which enable concurrent access to an unknown number of identical processes. Such data structures, denoted *public data structures*, are defined and studied in [BM94a], where the relation between wait-freedom (which guarantees that every operation eventually terminate), and bounded wait-freedom (which guarantees that every operation is terminated within a fixed and predetermined number of steps) in such data structure is studied.

In [FMRT90], Fischer, Moran, Rudich and Taubenfeld investigated a deceptively simple problem called the wakeup problem. The goal is to design a protocol for n asynchronous identical processes in a shared memory environment such that at least one process eventually learns that at least τ processes have woken up and begun participating in the protocol. All the solutions for that problem, except one, use some implementation of a counter in order to count the awake processes. There is one solution however, the see-saw protocol, where the counting is done somehow in the local memory of the processes and it requires only two bits of shared memory which are used for communication. The see-saw protocol cannot tolerate even one faulty process, and it seems that the approach taken in designing it cannot be adopted to solve the concurrent counter problem.

As mentioned earlier, when two or more identical processes may increment the counter, the read-modify-write atomicity, assumed in this paper, cannot be replaced by the weaker read/write atomicity. However, it is shown in [AH90a, AH90b, AG91, Lam77], and follows from the algorithm in Subsection 4.1, that it is possible to implement a counter using only read/write atomicity, when the processes have unique identifiers. In these implementations the basic correctness condition is *linearizability.* That is, although operations of concurrent processes may overlap, it should provide the illusion that each counter operation is atomic, while preserving the order in which operations that do not overlap happen.

In the paper we also cover the notion of a *linearizable counter*, which intuitively is a dynamic counter in which the executions of the `increment` and `look` operations are linearizable (see [HW90]). The *static*, *dynamic*, and *linearizable* counters bear some similarity to the notions of *safe*, *regular* and *atomic* registers defined by Lamport in [Lam86]. In a *safe* register, it is assumed only that a read not concurrent with any writes obtains the correct value. A *regular* register is a safe register in which a read that overlaps a write obtains either the old or new value. An *atomic* register, is a safe register in which the reads and writes behave as if they occur in some linear order.

2 Definitions and Notations

On a first reading of the paper, the reader may wish to skip this section and proceed immediately to the algorithms and theorems in later sections.

In this section we characterize asynchronous shared memory systems which support an atomic read-modify-write operation, and formally define static, dynamic and linearizable counter protocols. We start with a formal description of a protocol.

2.1 Protocols

A protocol $P = (C, N, R)$ consists of a nonempty set C of runs, a (possibly infinite) tuple $N = (p_1, p_2, \ldots)$ of processes and a tuple $R = (R_1, R_2, \ldots)$ of sets of registers. We may think of R_i as the set of all the registers that process p_i can access. A *run* is a pair (f, S) where f is a function which assigns initial values to the registers and S is a finite or infinite sequence of events. When S is finite, we also say that the run is finite.

An *event* corresponds to an atomic step performed by a process. Here we consider only the following types of events:

- $rmw_p(r, v, v')$ – process p first reads a value v from r and then writes a value v', that can depend on v, into r. This event is called read-modify-write.
- $begin\_look_p$ – process p starts a `look` operation.
- $end\_look_p(v)$ – process p ends a `look` operation and returns the value v (as the value of the counter).
- $begin\_inc_p$ – process p starts an `increment` operation.
- end_inc_p – process p ends an `increment` operation.

We use the notation e_p to denote an instance of an arbitrary event, which may be an instance of any of the above types of events, and say that e_p *involves* process p. (The subscript p is omitted when it is unimportant.)

The *value* of a register at a finite run is the last value that was written into that register, or its initial value (determined by f) if no process wrote into the register. We use $value(r, x)$ to denote the value of register r at a finite run x. A register r is said to be *local* to process p_i if $r \in R_i$ and for any $j \neq i, r \notin R_j$. A register is *shared* if it is not local to any process.

Let $x = (f, S)$ and $x' = (f', S')$ be runs. Run x' is a prefix of x (and x is an extension of x'), denoted $x' \leq x$, if S' is a prefix of S and $f = f'$. Let $\langle S; T\rangle$ be the sequence obtained by concatenating the finite sequence S and the sequence T. Then $\langle x; T\rangle$ is an abbreviation for $(f, \langle S; T\rangle)$. When $x' \leq x$, $(x - x')$ is the suffix of S obtained by removing S' from S.

For any sequence S, let S_p be the subsequence of S containing all events in S which involve p. Run (f, S) *includes* (f', S') iff $f = f'$ and S'_p is a prefix of S_p for all $p \in N$. Runs (f, S) and (f', S') are *indistinguishable* to the set of processes $\mathcal{G}$, denoted by $(f, S)[\mathcal{G}](f', S')$, iff $S_p = S'_p$ for every $p \in \mathcal{G}$, and $f(r) = f'(r)$ for every local register r of every process in $\mathcal{G}$. The relation $[\mathcal{G}]$ is an equivalence relation. When $\mathcal{G} = \{p\}$ we write $[p]$ instead of $[\mathcal{G}]$.

We assume throughout this paper that x is a run of a protocol if and only if all finite prefixes of x are runs. Notice that, by this assumption, if (f, S) is a run then $(f, null)$ is also a run, where *null* is the empty sequence.

2.2 Read-modify-write Protocols

Next, we characterize asynchronous shared memory systems by axioms that any protocol operating in such systems satisfies. We mention below only the axioms that are needed to prove the results. The axioms do not give a complete characterization of these systems.

DEFINITION **1** *An asynchronous read-modify-write protocol is a protocol in which the shared registers are accessed only by read-modify-write events whose runs satisfy axioms RMW1 – RMW3.*

AXIOMS FOR READ-MODIFY-WRITE

RMW1 Let $\langle x; rmw_p(r, v, v')\rangle$ and y be finite runs where $x[p]y$ and $value(r, x) = value(r, y)$. Then $\langle y; rmw_p(r, v, v')\rangle$ is a run.

RMW2 Let $\langle x; rmw_p(r, v, v')\rangle$ and y be finite runs where $x[p]y$. Then $\langle y; rmw_p(r, u, u')\rangle$ is a run for some values u and u'.

RMW3 Let $\langle x; rmw_p(r, v, v')\rangle$ be a run. Then $v = value(r, x)$.

RMW1 means that if a read-modify-write event which involves p can happen at a run, then the same event can happen at any run that is indistinguishable to p, provided that the register p accesses in that event has the same value in both runs. *RMW2* means that if a read-modify-write event which involves p can happen at a run, then some read-modify-write event in which p accesses the same register can happen at any run that is indistinguishable to p. *RMW3* means that it is possible to read only the last value that is written into a register. For the rest of the paper, unless otherwise stated, we consider read-modify-write protocols. Hence whenever we write *protocol* we mean *read-modify-write protocol.*

2.3 Counter Protocols

A counter (mod$m$) is a collection of shared registers $r_1, ..., r_\ell$ and some function *val* that associates some integer value – the value of the counter – in the range $\{0, ..., m-1\}$ to any possible assignment to the register. Let V_i denote the set of all possible values of register r_i, then the function *val* is from $V_1 \times \cdots \times V_\ell$ into $\{0, ..., m-1\}$.

A counter protocol is a protocol which supports two (non-atomic) operations on the counter: `increment` – in which some process increments the value of the counter by 1 (mod m), and `look` – in which some process reads the value of the counter. Each of these operations is a sequence of atomic (read-modify-write) operations applied to both local and shared registers. With each run x of a counter protocol we associate a value $count(x)$ defined by: $count(x) = val(value(x, r_1), ..., value(x, r_l))$, where x is some run and $r_1, ..., r_\ell$ are the counter registers. That is: $count(x)$ is the value assigned to the counter in x.

We say that a process p is *involved* in an `increment` operation in a given run if the number of $begin\_inc_p$ events in this run is strictly greater than the number of end_inc_p. We define similarly when a process is *involved* in a `look` operation. A process that is not involved in a `look` or `increment` operation is *idle* in the run. In other words, a process is idle in a given run if it had completed all the `increment` and `look` operations that it initiated during that run. If in a run two processes are both involved in `look` or `increment` operations, we say that the corresponding (executions of the) operations *overlap* each other in that run. A process is involved in a `look` or `increment` operation *during* a run, if it is involved in that operation in some prefix of that run.

DEFINITION **2** *A counter protocol is a protocol over an infinite set of processes whose runs satisfy axioms C1 – C6.*

THE COUNTER AXIOMS

C1 Let $\langle x; e_p\rangle$ and y be finite runs where $x[p]y$ and e_p is either $begin\_inc_p$, end_inc_p, $begin\_look_p$ or $end\_look_p(v)$. Then $\langle y; e_p\rangle$ is a run.

C2 For any run x and process p, $\langle x; rmw_p(r, v, v')\rangle$ is a run only if p is not idle in x.

C3 For any run x and process p, $\langle x; end\_look_p\rangle$ is a run only if p is involved in a `look` operation in x. Similarly, for any run x and process p, $\langle x; end\_inc_p\rangle$ is a run only if p is involved in an `increment` operation in x.

C4 If some process is activated infinitely many times in a certain run, then every $begin\_look_p$ [$begin\_inc_p$] event is followed by $end\_look_p(v)$ [end_inc_p] (i.e. every `look` or `increment` operation initiated by this process in this run is eventually terminated).

C5 For any run x and process p, $\langle x; begin\_inc_p\rangle$ is a run if and only if p is idle in x, and $\langle x; begin\_look_p\rangle$ is a run if and only if p is idle in x.

C6 Let $\langle x; begin\_look_p; S; end\_look_p(v)\rangle$ be a run, where S does not include any *rmw* step by a process which is involved in an `increment` operation. Then $v = count(x)$.

C1 means that if the event of beginning or ending an `increment` or `look` operation which involves p can happen at a run, then the same event can happen at any run that is indistinguishable to p. *C2* is a technical axiom and means that only `look` and `increment` operations are allowed. *C3* means that a process can complete a `look` or `increment` operation only if it started and has not completed the operation yet. *C4* is a wait-freedom requirement, which means that a process which is activated sufficiently many times must complete performing `look` or `increment` operations even if all other processes (including those which are involved in `increment` or `look` operations) have crashed. *C5* means that a process can start a `look` or an `increment` operation if and only if it is idle. *C6* means that when a run x is extended by a `look` operation which is not affected by a *rmw* step of an `increment` operation, the value returned by the `look` is $count(x)$.

2.4 Static, Dynamic and Linearizable Counter Protocols

We would like the `look` operation to return values that reflect, in some precise sense, an "actual" value of the counter at the time the `look` operation was executed. Intuitively, such a value equals the initial value of the counter plus the number of times the counter was incremented so far. There are three definitions of counters depending on the correctness requirements for the `look` operation: (1) *Static* counter – which requires that a `look` operation returns a correct value only if it does not overlap any `increment` operation. (2) *Dynamic* counter – which requires that it returns a correct value in all cases. (3) *Linearizable* counter – which is a dynamic counter in which the executions of the `increment` and `look` operations are linearizable. That is, it behaves as if each counter operation is atomic.

An *n-run* is a run in which at most n distinct processes execute `increment` operations (each such process may execute an arbitrary number of `increment` operations during the run). An ∞*-run* is a run in which there is no bound on the number of processes that execute `increment` operations. For the rest of the paper, $\oplus$ denotes addition modulo m.

Let x be a run. Then $begin(x)$ is the number of `increment` operations started in x (each of which may or may not have terminated yet), and $end(x)$ is the number of `increment` operations terminated in x. Clearly, for any run x, $end(x) \leq begin(x)$.

Definition **3** *A static n-counter protocol is a counter protocol whose n-runs satisfy axiom S.*

The static counter Axiom

S Let $x = (f, S)$ be a n-run where $begin(x) = end(x)$. Then $count(x) = count((f, null)) \oplus end(x)$.

The static counter axiom means that in every n-run of the protocol, every execution of a `look` operation which does not overlap any execution of an `increment` operation, returns the value $i \oplus s$ where s is the number of `increment` operations performed during the run, and i the initial value of the counter.

In defining dynamic counters, we use the notation $[a, c]$ to denote the set of values in $\{0, ...m-1\}$ that lie on the circle between a and c, inclusive, i.e.,

$$[a, c] = \begin{cases} \{b : a \leq b \leq c\} \cap \{0, ...m-1\} & \text{if } a \leq c; \\ (\{b : a \leq b\} \cup \{b : b \leq c\}) \cap \{0, ..., m-1\} & \text{if } a > c. \end{cases}$$

Definition **4** *A dynamic n-counter protocol is a counter protocol whose n-runs satisfy axiom D.*

The dynamic counter Axiom

D Let $x = (f, S)$ be a n-run, $z = \langle x; begin\_look_p; S'; end\_look_p(v)\rangle$ be a n-run which extends x, where S' does not include any $begin\_look_p$ event. Let $l = end(x) \oplus count((f, null))$, and $r = begin(z) \oplus count((f, null))$. If $begin(z) - end(x) < m$, then $v \in [l, r]$ (otherwise, v can be any value in $\{0, ..., m-1\}$).

The dynamic counter axiom means that in every n-run of the protocol, every execution of a `look` operation must return a value between the initial value plus the number of `increment` operations that were *completed* before the corresponding execution of the `look` operation was initiated, and the initial value plus the number of `increment` operations that were *initiated* before that execution was terminated.

Following the definition of linearizability introduced in [HW90], we now formally define linearizable counters, using the following notations. A *sequential* run is a run where at most one process is involved in an (`increment` or `look`) operation in any prefix of it. We use the notation $A \rightarrow B$ in x to mean that operation A has ended before operation B has started, in x. Notice that in any run that satisfies axiom $C5$, for any two operations A and B by the same process in that run, either $A \rightarrow B$ or $B \rightarrow A$.

Definition **5** *A linearizable n-counter protocol is a dynamic counter protocol whose n-runs satisfy axiom L.*

The linearizable counter Axiom

L For any n-run $x = (f, S)$ there exists a sequential n-run $y = (f, S')$ and a mapping *map* such that:

1. *map* is a 1-1 and onto mapping from the `increment` and `look` operations in x to these of y, s.t. an `increment` [`look`] operation of process p in x is mapped to a `increment` [`look`] operation of process p in y;
2. If $A \rightarrow B$ in x then $map(A) \rightarrow map(B)$ in y;
3. For any process p, the sequences of values returned by the `look` operations of p in x and in y are the same.

The linearizable counter axiom means that in every n-run of the protocol, an outside observer gets the illusion that every `look` and `increment` operation is atomic.

We also consider ℓ-*bounded* dynamic n-counters, which are counters which are dynamic correct only in runs with at most ℓ increment operations. Formally, these are n-counter which satisfy axiom D only when the run z in Axiom D satisfies $begin(z) \leq \ell$.

3 An Optimal Static ∞-Counter

In this section we present a static ∞-counter protocol that uses only $\log m$ read/write bits. That is, in the protocol there is no bound on the number of processes that are allowed to increment the counter and it matches the trivial $\log m$ bits lower bound. (We point out that it is impossible to solve the problem, when two or more precesses may increment the counter, using only atomic read and atomic write operations. In that case, it is possible to find two processes that behave the same when each of them run alone, and as already pointed out in the introduction, we can construct a run in which one of the processes is "hidden", by running the processes in lock steps.)

Theorem 3.1 *There is a static ∞-counter protocol which uses* $\log m$ *registers.*

In order to prove the theorem we describe the Positional Protocol. In this protocol a process may change the value of several registers during a single `increment` operation. In section 6, we show that any optimal static ∞-counter must allow processes to change the value of more than one register during a single `increment` operation, and that there is no dynamic ∞-counter protocol which achieve the same $\log m$ space complexity. We point that the protocol is not even dynamic 1-counter protocol.

3.1 The Positional Protocol

In the Positional Protocol the contents of the $k = \log m$ shared registers $r_{k-1}, \cdots, r_0$ are viewed as a binary representation of the value of the counter. That is, the binary word $\overline{v} = [v_{k-1} \cdots v_0]$ represent the value $binary(\overline{v}) = \sum_{i=0}^{k-1} 2^i$. The `increment` operation is performed by the straightforward (sequential) algorithm for incrementing a binary number. That is: scan the the registers from right to left (starting with r_0); when scanning register r_i, do the following: (1) flip r_i, and (2) if the value of r_i was 1 before it was flipped and $i < k-1$, then repeat this operation on register r_{i+1}, else terminate the `increment` operation. The `look` operation is performed by simply reading the contents of the registers $\overline{v}$ and then converting it to the appropriate value, $binary(\overline{v})$.

The correctness proof of this simple implementation is somewhat complicated by the fact that several `increment` operations may take place simultaneously. The proof is based on showing that in any execution in which k complete `increment` operation are performed (for arbitrary k) and no other increment is initiated, the number of times each register is changed depends only on the initial contents of the shared registers and on k, regardless the order by which the registers were accessed by the various processes.

In Figure 1 the code of the `increment` and `look` operations is given. In order to describe the read-modify-write operations, we introduce the special purpose constructs *lock(r)* and *unlock* into the programming language. These constructs mark the beginning and end of atomic, exclusive access to a shared register r. It is assumed that a process can lock only one register at a time, that a process does not fail between pairs of *lock*(r) and *unlock* statements, and that any non-faulty process that reaches a *lock* statement eventually executes it. This corresponds to the assumption in our underlying model that a read-modify-write operation is wait-free and atomic.

3.2 Correctness Proof

We will use the following notions. A run is a *serial increment run* of a counter protocol if in any prefix of it, at most one process is involved in an `increment` operation. A *complete run* is a run in which no process is involved in an `increment` operation. Two complete runs are *similar* if the initial contents of the counter registers are the same in both runs, and the same number of `increment` operations were performed in both runs. Two complete

```
function **look**;
begin
    /* read the values of the registers */
    for $i := 0$ to $k-1$ do $a[i] := r_i$;
    return $binary(a[k-1], \cdots, a[0])$;
end;

procedure **increment**;
begin
    $i := 0$;
    repeat
        /* execute a read-modify-write instruction on $r_i$ */
        lock($r_i$)
            $flag := r_i$;
            $r_i := r_i + 1 \pmod 2$;
        unlock;
            $i := i + 1$;
    until $flag = 0$ or $i = k$;
end;
```

Figure 1: The Positional Protocol

similar runs are *equivalent* if they have the same final value of the counter. Recall that for a complete run x, $end(x)$ denotes the number of (completed) `increment` operations performed in x.

First we show that any two similar complete runs of the positional protocol are equivalent. For this we show that the number of times a register is changed during any complete run depends only on the initial contents of the shared registers and on the number of `increment` operations performed. Then, we observe that the protocol is correct when we consider only serial increment runs. Since any complete run is similar to some complete serial increment run, it follows that any run is also equivalent to some complete serial increment run, which implies that the protocol is correct for all complete runs.

Lemma 3.1 *Let x be a complete run of the Positional Protocol, let $\widehat{r_i}$ be the initial contents of the counter register r_i ($0 \le i < k$), and let $C(r_i, x)$ be the number of times r_i was flipped in x. Then,*

$$C(r_i, x) = \begin{cases} end(x) & \textit{if } \; i = 0; \\ \left\lfloor \frac{C(r_{i-1}, x) + \widehat{r_{i-1}}}{2} \right\rfloor & \textit{otherwise.} \end{cases}$$

Proof: The proof is by induction on r_i. By observing the algorithm in figure 1 it is immediate that each execution of an `increment` operation changes r_0 (the rightmost register) exactly once. Thus, r_0 is flipped $end(x)$ times.

Suppose the lemma holds for register r_{i-1}, we show that it also holds for register r_i. If $\widehat{r_{i-1}}$ is 0, then r_{i-1} was changed $\left\lfloor \frac{C(r_{i-1},x)}{2} \right\rfloor$ times from 1 to 0, and if $\widehat{r_{i-1}}$ is 1, then r_{i-1} was changed $\left\lfloor \frac{C(r_{i-1},x)+1}{2} \right\rfloor$ times from 1 to 0. Every complete execution of an `increment` operation that changes r_{i-1} from 1 to 0 changes also r_i, and every such execution that changes r_{i-1} from 0 to 1 halts. Hence the number of times r_i was changed is $\left\lfloor \frac{C(r_{i-1},x)+\widehat{r_{i-1}}}{2} \right\rfloor$. ▮

Lemma 3.2 *Let x be a complete run of the Positional Protocol, let $\overline{v}$ be the initial contents of the shared registers, and let $\overline{w}$ be the final contents of the shared registers (i.e., when x terminates). Then $binary(\overline{w}) = binary(\overline{v}) \oplus end(x)$.*

Proof: It follows immediately from the properties of the (sequential) algorithm for binary addition that $binary(\overline{w}) = binary(\overline{v}) \oplus end(x)$ for any complete serial increment run. It is shown in Lemma 3.1 that the number of times a register is changed during any complete run depends only on the initial contents of the shared registers and on the number of `increment` operations performed. This implies that any two similar complete runs of the positional protocol are equivalent. Since any complete run is similar to some complete serial increment run, it follows that any run is also equivalent to some complete serial run. Thus, since the lemma holds for all complete serial runs, it holds for all complete runs. ▮

Theorem 3.2 *The Positional Protocol is a static ∞-counter protocol.*

Proof: We have to show that for any run, if a process executes a `look` operation which does not overlap any execution of an `increment` operation, the value that it returns is the initial value of the counter plus the number of `increment` operations performed. We use the following two trivial observations. The implementation of the `look` operation does not change the value of the registers and when it does not overlap `increment` operation it return the integer whose binary representation is the actual value of the counter registers. The fact that a `look` operation does not overlap any `increment` operation means that the run which terminates just before the start of that `look` operation is a complete run. By using Lemma 3.1 again, all these imply that the `look` operation returns the right value. ▮

4 An Optimal Dynamic 1-Counter

In the previous section we presented a space optimal static ∞-counter protocol. The situation with dynamic counters is considerably more involved, and in the general case dynamic counters will require much more space than static ones. One specific case where we are able to design a dynamic counter that matches the trivial $\log m$ bits lower bound is the case where only one process can increment the counter.

Theorem 4.1 *There is a dynamic 1-counter protocol that uses $\log m$ bits.*

To prove the theorem we present a protocol, where each `increment` operation changes the value of exactly one bit of the counter, and a `look` operation never changes the value of the counter. The motivation for this protocol is due to the following lemma. A counter protocol is *1-flip counter*, if a process must change the value of exactly one bit during a single `increment` operation, and no bit can be flipped during a `look` operation.

Proposition 4.1 *Any dynamic 1-counter protocol that uses* $\log m$ *bits must be 1-flip counter.*

Proof: Let $m = 2^k$, and let Pr be a dynamic 1-counter protocol which uses $\log m$ registers. It is sufficient to show that for every complete run ρ of Pr, and for every extension ρ' of ρ where $(\rho' - \rho)$ contains exactly one complete `increment` operation, the value of exactly one register is changed in $(\rho' - \rho)$.

Let p be the process that increments the counter. Consider a run that starts when the contents of the counter is some k-bit word u_0, assume w.l.o.g. that $val(u_0) = 0$, and proceeds as follows: Initially, a complete `look` operation is performed (by some process), then a complete `increment` operation is performed by p, then another complete `look` operation is performed, and so on until $m - 1$ `increment` operations and m `look` operations are performed.

Let u_i be the contents of the counter registers immediately after the i-th `increment` operation is completed. Then the following `look` operation must return the value $i = val(u_i)$. Thus the set $\{u_0, \cdots, u_{m-1}\}$ consist of $m = 2^k$ distinct words, s.t. $val(u_i) \neq val(u_j)$ when $i \neq j$. Since there are only m k-bits words, we conclude that for each pair of k-bit words u and v, $val(u) \neq val(v)$.

Now we use the above observation to show that for any run ρ, in any extension of ρ by a complete `look` operation no register is changed. Let u be the contents of the counter in ρ. Then a process executing a complete `look` operation cannot distinguish ρ from another complete run where the contents of the counter equals u. Assume that during the execution of the `look` operation the value of some register is changed, and let v be the contents of the counter registers after the first register was changed by the `look` operation. Since ρ is a complete run, by the correctness requirement it must be that $val(u) = val(v)$, a contradiction.

Next, to prove the lemma, assume to the contrary that there is an extension ρ' of ρ in which process p executes one complete `increment` operation in $(\rho' - \rho)$, and in $(\rho' - \rho)$ the values of at least two registers are changed.

Let the contents of the counter in ρ be u, and let r_i and r_j $(i \neq j)$ be the first and second registers to be changed in $(\rho' - \rho)$, respectively. Let v be the contents of the counter after the value of register r_i is changed, and let w be the contents of the counter after the value of register r_j is changed. Notice that u, v and w are distinct and hence as proved earlier, $val(u), val(v)$ and $val(w)$ must be distinct. However by the definition of dynamic counter, it must be that $val(u) \leq val(v) \leq val(u) + 1$, and $val(u) \leq val(w) \leq val(u) + 1$, and hence it can not be that all the three values are distinct, a contradiction. ∎

It is interesting to note that dynamic 1-flip counters are, in fact, linearizable counters, since we can order the `look` and `increment` operations of each execution in a complete order, which is consistent with the partial order defined by that execution. This can be done as follows:

1. First order the `increment` operations according to the order of the read-modify-write instruction that flipped the value of a register. Note that since this is a 1-flip protocol, this order is well defined.

2. Then, order each `look` operation which overlaps at least one `increment` operation, after one of the `increment` operations which overlap it, where the value of the counter immediately after that increment operation equals the value returned by the `look` operation. After this step, all the `increment` operations are ordered, and all the `look` operations are ordered relative to the `increment` operations.

3. Finally, order each set of `look` operations that appear between two consecutive `increment` operations in an arbitrary way.

4.1 Preliminaries

For many years, Gray code has been used for implementing counting [Gar72, Gil58, Gra53, Koh70]. In these implementations, increments are done, by only one incrementor, according to the code, while a `look` operation simply reads the counter digits, and converts them to the appropriate number. This technique works *only* if a `look` operation is much faster than an `increment` operation. In the case where few `increment` operations are concurrent with a `look` operation, the look might return a wrong answer. In our framework we do not assume anything about the relative speeds of `increment` and `look` operations, hence the above naive use of Gray code does not solve our problem.

Reflected binary Gray code (abbv. Gray code) is a well known method to order all binary words of any given length k in a cyclic order, such that two successive words differ in exactly one bit. It is called reflected code because it can be generated by the following simple algorithm. Start with 0,1 as a one-digit Gray code, then reflect and append the digits to get 0,1,1,0. Next put 0's in front of the first two numbers and 1's in front of the last two numbers. The result is a two-digit gray code 00,01,11,10. To extend an i-digit Gray code to an $(i+1)$-digit code, reflect the the i-digit code and, as before, put 0's in front of the first half of these numbers and 1's in front of the last half. Note that the resulting code is cyclic in that the first and last numbers also differ at only one position. A 4 bit Gray code is given in Figure 2. Let $G_k = (g_0, g_1, \cdots, g_{m-1})$ be all the binary words of length $k = \log m$ ordered by Gray code (where g_0 is the allzero word). Let *gray* be the 1-1 mapping defined by $gray(g_i) = i$. Our protocol represents each integer $i \in \{0, \cdots, m-1\}$ by g_i.

An important known property is that the mapping *gray* and its inverse $gray^{-1}$ are computable by an on-line linear time algorithm. To convert a standard binary number to its reflected Gray equivalent start with the digit at the right and consider each digit in turn. If the next digit to the left is 0, let the former digit stand. If the next digit to the left is 1, change the former digit. The leftmost digit is assumed to have 0 on its left and therefore remains unchanged. To convert back again consider each digit in turn starting at the right. If the parity (sum) of all digits to the left is even, let the digit stay as it is. If the parity is odd, change the digit.

There is also a known on-line linear time algorithm to find the successor of a word in G_k. For each k-bits word $v = [v_{k-1} \cdots v_0]$, define the *critical index* of v to be the minimal index j in $\{0, \cdots, k-1\}$ such that the $k - j$ bits prefix of v, $[v_{k-1} \cdots v_j]$, has even parity,

0	0000	
1	0001	
2	0011	
3	0010	
4	0110	$=first([01])$
5	0111	$=middle([01])$
6	0101	
7	0100	$=last([01])$
8	1100	
9	1101	
10	1111	
11	1110	
12	1010	
13	1011	
14	1001	
15	1000	

Figure 2: A 4 bit Gray code

and if there is no such index (i.e., $v = [10\cdots 0]$) then $j = k-1$. Then the successor of v in G_k is obtained from v by flipping the bit v_j, where j is the critical index of v.

The next property which we use is that for any word w where $|w| = i < k$, all the k-bits words whose prefix is w appear consecutively in G_k.

Using this property it is easy to prove the following useful property. Denote by $first(w)$ the first word in G_k that has w as a prefix. Similarly, denote by $last(w)$ the last word in G_k which has w as a prefix. Finally, $middle(w)$ is the the unique word v, for which $gray(v) = \frac{gray(first(w))+gray(last(w))}{2}$. Let $par(w)$ be the parity w. Then $first(w) = w \cdot par(w) \cdot 0^{k-i-1}$ (i.e., w followed by the parity bit of w followed by $k-i-1$ zeroes), $last(w) = w \cdot \neg par(w) \cdot 0^{k-i-1}$, and $middle(w) = w \cdot par(w) \cdot 1 \cdot 0^{k-i-2}$. See example in Figure 2. (The proof is by induction on the length of w and is left for the reader.)

4.2 The Gray Code Counter

We can now give a description of the *Gray code counter.* The code of the protocol is given in Figure 3.

For the rest of this section m and $k = \log m$ are fixed, and k is the number of registers used for the counter.

The `increment` operation is performed by reading the contents $v = [v_{k-1} \cdots v_0]$ of the counter registers, and flipping v_j, where j is the critical index of v described above. We notice that since there is only one process that may increment the counter, it has to read the counter only one time, at the beginning of the protocol.

The protocol for `look` uses a function called *4-way scan* whose purpose is to take a snapshot of the contents of the counter registers, and to return a word g_i such that i is a valid value of the counter. This function is described next. It first reads the registers from right to left and gets a word $a = [a_{k-1} \cdots a_0]$. That is, a_0 is read first and a_{k-1} is read last. Then it reads the registers from left to right and gets a word $b = [b_{k-1} \cdots b_0]$ (this time b_{k-1} is read first). Then, again, it reads the registers from right to left and gets a word $c = [c_{k-1} \cdots c_0]$, and, finally it reads the registers from left to right and gets a word $d = [d_{k-1} \cdots d_0]$.

Let w be the maximal common prefix of the words a, b, c and d, and let $|w| = i$. In case $i < k$, let x_1, x_2, x_3 and x_4 be the $i + 1$st bit of a, b, c and d respectively. (I.e., $w \cdot x_1$ is a prefix of a.) The 4-way scan function checks the following conditions sequentially:

The 4-way scan function:

1. if $a = b$ then return a;
2. elseif $c = d$ then return c;
3. elseif $|w| = 0$ then return $last([a_{k-1}])$;
4. elseif $x_1 = \neg par(w)$ and $x_2 = x_3 = x_4 = par(w)$ then return $first(w)$;
5. elseif $x_1 = x_2 = x_3 = \neg par(w)$ and $x_4 = par(w)$ then return $last(w)$;
6. elseif $x_1 = x_2 = \neg par(w)$ and $x_3 = x_4 = par(w)$ then return $last(w)$;
7. elseif $x_1 = x_3 = \neg par(w)$ and $x_2 = x_4 = par(w)$ then return $last(w)$.
8. elseif all the above conditions fail then return $middle(w)$.

Using the *4-way scan*, it is now easy to describe the implementations of the `look` operation. The `look` operation calls the *4-way scan* function that returns a k bit word, g. The output of the `look` operation is $gray(g)$.

```
0:  function **look**;
1:  begin
2:        return(gray (4-way scan));
3:  end;

0:  procedure **increment**;
1:  begin
2:        $v$ := current contents of the counter registers;
3:        $j$ := the critical index of $v$;
4:        $r_j := \neg(r_j)$;
5:  end;
```

Figure 3: The Gray code counter

4.3 Correctness Proof

In order to prove the correctness of the Gray code counter protocol, it is sufficient to consider only runs ρ of the following type: The first atomic step in ρ is the first atomic step of some `look` operation, the last atomic step in ρ is the last step of the same `look` operation, and all other steps in ρ (if there are any) are not related to any other `look` operations.

For the rest of the section we will assume only such runs with a single `look` operation. Recall that the `look` operation uses a 4-way scan which scan the counter registers 4 times.

Let ρ be a run, or a subsequence of consecutive steps in a run. In the proof we use the concept $valid(\rho)$, which is the cyclic interval containing the counter values during ρ. Formally, if at least m increment operations have occurred in ρ then $valid(\rho) = \{0, ..., m-1\}$ otherwise it includes all the values in the cyclic interval $[gray(u), gray(v)]$, where u and v are the initial and final contents of the counter in ρ, respectively. (The notion of a cyclic interval $[a, b]$ is defined in section 2.) Using the above notation, we have to show that in each run ρ which consists of a single `look` operation as described above, the `look` operation returns a value in $valid(\rho)$.

Let w be an i-bit word for $i \leq k$. The *block of* w of words of size k, denoted by $block_k(w)$ is the set $\{w \cdot u : u \text{ is a } (k-i)\text{-bits word}\}$. An *$i$-block* is $block_k(w)$ for some k and some i-bit word w. Thus, two words are in the same i-block if they are of the same length and have the same prefix of length i. For example the words [1011] and [1001] are in the same 1-block and 2-block, but are not in the same 3-block. As already pointed out, for each i-bits word w all the words in $block_k(w)$ are consecutive in G_k. Also, we notice that $block_k(w)$ in G_k, consists of two consecutive $(i+1)$-blocks. We will omit the subscript k in $block_k(w)$ when it can be understood form the context.

Lemma 4.1 *Let ρ be a subsequence of consecutive steps in a run which consists of either the first and second scans or in the third and fourth scans of the 4-way scan function, and let a and b be the corresponding two words read in ρ. Assume both a and b are in $block(w)$ for some i-bit word w. Then,*

1. *$valid(\rho)$ contains a word in $block(w)$.*
2. *Assume that a is in the first $(i+1)$-block of $block(w)$ and b is in the second $(i+1)$-block of $block(w)$. Then $valid(\rho)$ contains the last word in the first $(i+1)$-block and the first word in the second $(i+1)$-block of $block(w)$.*

Proof:
1. We prove the first part of this lemma by proving a stronger claim: For $i = 1, \cdots, k$ let u_i $[v_i]$ be the contents of the counter while the bit $a(k-i)$ $[b(k-i)$ resp.$]$ was read, and let $valid_i(\rho)$ be the cyclic interval $[gray(u_i), gray(v_i)]$. (Thus, $valid_1(\rho) \subseteq valid_2(\rho) \ldots \subseteq valid_k(\rho) = valid(\rho)$.) We prove by induction on i that if both a and b are in $block(w)$ for some i-bit word w, then $valid_i(\rho)$ contains a word in $block(w)$.

The base of the induction is for $i = 1$. The correctness here is implied by the fact that if $a(k-1) = b(k-1) = b$ then by definition both u_1 and v_1 are in $block([b])$.

Let us suppose the claim is true for a and b which are in the same $(i-1)$-block, and let w be an i-bit word. We will show that if a and b are also in $block(w)$, then the claim still holds.

Let a and b both be in $block(w)$, and let u be the prefix of the first $(i-1)$ bits of w. Then both a and b are in $block(u)$, and hence, by induction, $valid_{i-1}(\rho)$ contains a word in $block(u)$. We prove that $valid_i(\rho)$ must contain a word of $block(w)$.

There are two possible cases. The first one is the case where $w = u \cdot [par(u)]$, meaning that $block(w)$ is the first i-block of $block(u)$. If $valid_{i-1}(\rho)$ contains a word in $block(w)$ then we are done, since $valid_{i-1}(\rho) \subseteq valid_i(\rho)$. So assume that this is not the case, and hence $valid_{i-1}(\rho)$ contains a word v in the second i-block of $block(u)$. Since $v \in valid_i(\rho)$ we have that $valid_i(\rho)$ contains the cyclic interval $I = [gray(u_i), gray(v)]$. Since $a(k-i) = par(u)$, and a_{k-i} is the $k-i$'th bit of u_i, u_i is not in the second i-block of u. By the said above, v is in the second i-block of u. This means that I must contain a word in the first i-block of u, which is $block(w)$.

The second case is where $w = u \cdot [\neg par(u)]$, meaning that w is the second i-block of $block(u)$. If $valid_{i-1}(\rho)$ contains a word in $block(w)$ then we are done, so assume that this is not the case. This implies that $valid_{i-1}(\rho)$ contains a word v in the first i-block of $block(u)$. Since $valid_{i-1}(\rho) \subseteq valid_i(\rho)$, $v \in valid_i(\rho)$. In particular, $valid_i(\rho)$ contains the cyclic interval $I = [gray(v), gray(v_i)]$, where v is in the first i-block of u and v_i is not in this block. This means that I must contains a word in the second i-block of u, which is $block(w)$.

2. We prove the second part of this lemma using the first part. For simplicity, we will assume without loss of generality that $par(w) = 0$, therefore the first $(i+1)$-block in $block(w)$ is $block(w \cdot [0])$ and the second $(i+1)$-block is $block(w \cdot [1])$ (i.e $a(k-i-1) = 0$ and $b(k-i-1) = 1$).

By the first part of the lemma $valid_i(\rho)$ contains a word v in $block(w)$, hence $valid_{i+1}(\rho)$ also contains v. Since $u(k-i-1) = 0$, $u_i \notin block(w \cdot [1])$. This implies that $valid_{i+1}(\rho)$ contains a cyclic interval $I_1 = [gray(u_i), gray(v)]$ where u_i is not in $block(w \cdot [1])$ and v is in $block(w)$. Since $block(w \cdot [0])$ immediately precedes $block(w \cdot [1])$, I_1 must contain a word in $block(w \cdot [0])$. By a similar argument, $valid_{i+1}(\rho)$ contains a cyclic interval $I_2 = [gray(v), gray(v_i)]$, where $v \in block(w)$ and $v_i \notin block(w \cdot [0])$, thus I_2 must contain a word in $block(w \cdot [1])$.

Finally, observe that $valid_{i+1}(\rho)$ is actually combined of the cyclic segment I_1 followed by the cyclic segment I_2. Since I_1 contains a word in $block(w \cdot [0])$ and I_2 contains a word in $block(w \cdot [1])$, $valid_{i+1}(\rho)$ must contain the last word in $block(w \cdot [0])$ and the first word in $block(w \cdot [1])$. ∎

Theorem 4.2 *The Gray code counter is a dynamic 1-counter.*

Proof: We must show that for any run ρ with a single `look` operation, the value returned by the `look` operation in ρ is in $valid(\rho)$. In the 4-way scan function eight conditions are checked. We will prove correctness for each of these conditions. Assume without loss of generality that $par(w) = 0$ (i.e., the first $(i+1)$-block of $block(w)$ is $block(w \cdot [0])$ and the second $(i+1)$-block of $block(w)$ is $block(w \cdot [1])$).

We use a, b, c and d to be the vectors that are read in the first, second, third and forth scans of the 4-way scan, respectively. Let α be the prefix of ρ that ends at the point where b is read, and let β be $(\rho - \beta)$.

1. The condition checked is $a = b$. The first part of Lemma 4.1 proves that in this case $gray(a)$ is a valid value.

2. The condition checked is $c = d$. The first part of Lemma 4.1 proves that in this case $gray(c)$ is a valid value.

3. This is the case where the four words are not in the same 1-block. This means that the leftmost bit of the counter was changed sometime during the reading of the four words. Hence the integer value represented by the last word in the 1-block of a is in $valid(\rho)$.

4. Here we check the case where a is in $block(w \cdot [1])$ and b, c and d are in $block(w \cdot [0])$. From Lemma 4.1 and the fact that both c and d are in $block(w \cdot [0])$, we know that there exists a word $v \in block(w \cdot [0])$ which is in $valid(\beta)$. By an argument similar to the one used to prove Lemma 4.1, since the $i+1$ block containing a is not $block(w \cdot [0])$, the integer value represented by the first word in $block(w \cdot [0])$ (which is $gray(first(w))$) is a valid value.

5. In this case d is in $block(w \cdot [0])$ and a, b and c are in $block(w \cdot [1])$. From Lemma 4.1 and the fact that both a and b are in $block(w \cdot [1])$, we know that there exists a word $v \in block(w \cdot [1])$ which is in $valid(\alpha)$. Since the $i+1$ block containing d is not $block(w \cdot [1])$, the integer value represented by the last word in $block(w \cdot [1])$ (which is $gray(last(w))$) is a valid value.

6. This is the case where both a and b are in $block(w \cdot [1])$, and both c and d are in $block(w \cdot [0])$, From Lemma 4.1 we know that there exists a word $v \in block(w \cdot [1])$ which is in $valid(\alpha)$, and there exists a word $u \in block(w \cdot [0])$ which is in $valid(\beta)$. Since all the words in the cyclic interval starting with $valid(\alpha)$ and ending with $valid(\beta)$ are valid, the integer value represented by the last word of $block(w \cdot [1])$, which is the last word of the i-block, $gray(last(w)))$, is valid. (In fact, also the integer value represented by the first word of $block(w \cdot [0])$ $(i+1)$-block which is the first word in the i-block, $gray(first(w))$ is valid).

7. This is the case where both a and c are in $block(w \cdot [1])$, and both b and d are in $block(w \cdot [0])$. Denote the first words in $valid_{k-i-1}(\alpha)$ and $valid_{k-i-1}(\beta)$ by u_α and u_β respectively, and the last words in $valid_{k-i-1}(\alpha)$ and $valid_{k-i-1}(\beta)$ by v_α and v_β respectively.
   Case 1: $valid_{k-i-1}(\alpha)$ contains a word in $block(w \cdot [1])$. Then, since $u_\beta \notin block(w \cdot [1])$, the integer value of the last word in $block(w)$, $gray(last(w))$, is a valid value.
   Case 2: Not Case 1, i.e. $valid_{k-i-1}(\alpha)$ contains a word in $block(w \cdot [0])$. Then, since $v_\alpha \notin block(w \cdot [0])$ and $u_\beta \notin block(w \cdot [1])$, it must be the case that $gray(last(w))$ is a valid value.
   (In this case, it can also be shown that $grey(first(w))$ is a valid value).

8. If none of the above conditions were met, all the remaining possibilities can be grouped into two cases:

   (a) $x_1 = par(w)$ (which we assumed without loss of generality to be 0)

   (b) $x_4 = \neg par(w)$

Suppose that condition (a) is true. Let u_1 be the first word in $valid_{k-i-1}(\alpha)$ ($u_1 \notin block(w \cdot [1])$). From Lemma 4.1 we know that $valid_{k-i-1}(\alpha)$ includes a word u_2 from $block(w)$. If $u_2 \in block(w \cdot [1])$ then $middle(w)$ is in $valid_{k-i-1}(\alpha)$. Else there exists a word u_3 in $valid_{k-i-1}(\alpha)$ or $valid_{k-i-1}(\beta)$ such that $u_3 \notin block(w \cdot [0])$ (or else w would not be a maximal prefix), therefore $middle(w)$ is either in $valid_{k-i-1}(\alpha)$, $valid_{k-i-1}(\beta)$ or in the range between them, and $gray(middle(w))$ is a valid value.

The proof of the second case is similar.

∎

5 Dynamic Counters for Many Processes

In this section we present two dynamic counters. The first one, called the *cyclic flip* counter, works when the number of `increment` operations is bounded by m and requires m bits. The other is a combination of the Gray-code protocol and the cyclic flip protocol, and it works when the number of processes is bounded and known, but the number of `increment` operations may be unbounded.

5.1 The Cyclic-flip Counter

In this subsection we present an m-bounded dynamic ∞-counter protocol – that is, a protocol that works as long as no more than a total of m `increment` operations are performed. Counters of this type are used (implicitly) in wakeup and consensus protocols.

Theorem 5.1 *There is an m-bounded dynamic ∞-counter protocol that uses m registers.*

The protocol presented here, called the Cyclic-flip protocol, uses m registers, which is exponentially more than required by the dynamic 1-counter of the previous section. In the next section we show that an exponential gap in the number of registers is unavoidable, even if it is assumed that all the registered are initialized, and the processes are allowed to run distinct programs.

We point out that the space complexity of the Cyclic-flip protocol almost matches the $m-1$ lower bound which follows from Theorem 6.2 in the next section.

5.1.1 The Counting Method

Like the Positional and the Gray code counters, the Cyclic-flip counter is based on a mapping from binary words onto $\{0, 1, \cdots, m-1\}$, where m is a power of two. However, this time the domain of the mapping is the words of length m, and not of length $\log m$ as in the previous cases. The mapping, called *cyc*, is defined as follows:
Let w be a binary word of length m where m is a power of 2, and let $par(w)$ be the parity of w. When $m > 1$, let $w = w_2 \cdot w_1$ where $|w_2| = m/2$. Then,

$$cyc(w) = \begin{cases} 0 & \text{if } |w| = 1; \\ 2 \cdot cyc(w_2) + par(w) & \text{otherwise.} \end{cases}$$

That is, $cyc(w) = \sum_{i=0}^{\log m - 1} v(i) \cdot 2^i$, where $v(i) = par(w(m-1) \cdots w(m - 2^{\log m - i}))$. ($v(i)$ is the parity of the $\frac{m}{2^i}$ leftmost bits of w.) Thus, for m=8, $cyc(10110001) = 2^2 + 2^1 = 6$, since 10, 1011 have parity 1, and 10110001 has parity 0.

Given a word w, the `increment` operation is done by finding a word w' such that $cyc(w') = cyc(w) + 1$. For this we use the function *next*, defined below.

Let w be a binary word of length $m = 2^k$. When $m > 1$, let $w = w_2 \cdot w_1$ where $|w_1| = |w_2| = |w|/2$. Then,

$$next(w) = \begin{cases} [0] & \text{if } w = [1]; \\ [1] & \text{if } w = [0]; \\ w_2 \cdot next(w_1) & \text{if } par(w_1) = par(w_2); \\ next(w_2) \cdot w_1 & \text{if } par(w_1) \neq par(w_2). \end{cases}$$

We define $next^\ell(w)$ recursively as follows: $next^0(w) = w$ and for $\ell > 0$, $next^\ell(w) = next(next^{\ell-1}(w))$.

Lemma 5.1 *Let w be a binary word of length m. Then, $next^\ell(w)$ and w differ by exactly ℓ bits, for any $0 \le \ell \le m$.*

Proof: The proof is by induction on m, the length of w. The base of the induction is $m = 1$. In this case, by definition of the function *next*, $next^0([0]) = [0]$, $next^0([1]) = [1]$, $next^1([0]) = [1]$,and $next^1([1]) = [0]$. Hence, the lemma holds when $|w| = 1$.

Assume that the lemma holds for words of length $m/2$. We now prove that it also holds for words of length m. Let $w = w_2 \cdot w_1$, where $|w_1| = |w_2| = m/2$.

It is easily observed that for any $0 \le \ell \le m$, either $next^\ell(w) = next^{\lceil \ell/2 \rceil}(w_2) \cdot next^{\lfloor \ell/2 \rfloor}(w_1)$ or $next^\ell(w) = next^{\lfloor \ell/2 \rfloor}(w_2) \cdot next^{\lceil \ell/2 \rceil}(w_1)$. In any of the two cases, by applying the induction hypothesis twice [1] we get that $next^\ell(w)$ and w differ by exactly $\lceil \ell/2 \rceil + \lfloor \ell/2 \rfloor = \ell$ bits. ∎

Lemma 5.1 implies that the *next* function is a permutation which partitions the set of all binary words of length m into $2^m/2m$ cycles of length $2m$ each. Each such cycle consists of $2m$ words $w_0, w_1, \cdots, w_{2m-1}$, where $next(w_i) = w_{i+1 (\bmod\ 2m)}$. Furthermore, $w_{i+1(\bmod\ 2m)}$ is obtained from w_i by flipping one bit, and in any m successive applications of *next*, each bit is flipped exactly once.

The following lemma implies that when the value of the counter is given by the function *cyc*, the *next* function can be used for implementing the `increment` operation.

Lemma 5.2 *Let w be a binary word of length m. Then, $cyc(next(w)) = cyc(w) + 1 \pmod m$.*

Proof: The proof is by induction on m the length of w.
The base of the induction is $m = 1$. When $w = [i]$ where $i \in \{0, 1\}$, we have that,

$$cyc(next([i])) = cyc([1-i]) = 0 = cyc([i]) + 1 \pmod 1.$$

Assume that the claim holds for words of length $m/2$. That is, $cyc(next(w)) = cyc(w) + 1 \pmod{m/2}$, where w is a binary word of length $m/2$. We now prove that it also holds for words of length m. Let $w = w_2 \cdot w_1$, where $|w_1| = |w_2| = m/2$.

[1]Note that since m is even, $\ell \le m$ implies that $\lceil \ell/2 \rceil \le m/2$.

There are two possible cases.
Case 1: $cyc(w)$ is even. In this case, $cyc(w) = 2 \cdot cyc(w_2)$, and $par(w) = 0$. The fact that $par(w) = 0$ implies that $par(next(w)) = 1$, and that $next(w) = w_2 \cdot next(w_1)$. Thus,

$$cyc(next(w)) = cyc(w_2 \cdot next(w_1)) = 2 \cdot cyc(w_2) + par(next(w)) = cyc(w) + 1.$$

Case 2: $cyc(w)$ is odd. In this case, $cyc(w) = 2 \cdot cyc(w_2) + 1$, and $par(w) = 1$. The fact that $par(w) = 1$ implies that $par(next(w)) = 0$, and $next(w) = next(w_2) \cdot w_1$. Thus,

$$\begin{aligned} cyc(next(w)) &= cyc(next(w_2) \cdot w_1) = 2 \cdot cyc(next(w_2)) + par(next(w)) \\ &= 2 \cdot [cyc(w_2) + 1 \pmod{m/2}] = 2 \cdot (cyc(w_2) + 1) \pmod{m} \\ &= (2 \cdot cyc(w_2) + 1) + 1 \pmod{m} = cyc(w) + 1 \pmod{m}. \end{aligned}$$

∎

5.1.2 The Protocol

We are now ready to give a description of the *Cyclic-flip* protocol. The protocol uses m registers. The *val* function used by the protocol is the function *cyc* described above, which assigns a number in $\{0, \cdots, m-1\}$ to any given contents of the registers. The code for the Cyclic-flip protocol is given in figure 4. Below we give a description of its operations.

The `look` operation calls the *double-scan* function which reads all the m counter registers sequentially in a cyclic order, until it reads for two consecutive times the same values for all the registers, and output this m bit word. Then the *cyc* function is applied to the m bit output of the *double-scan* function and the result is some number v. The output of the `look` operation is the number v. The code for the *double-scan* function is given in figure 4.

The `increment` operation uses first the *double-scan* function to get a correct snapshot of the counter register. Then, the *next* function is used to find what bit has to be flipped in order to increment the counter. Finally if nobody else flipped this bit yet then this bit is flipped, otherwise we start all over again.

5.1.3 Correctness Proof

We now prove the correctness of the Cyclic-flip protocol.

Theorem 5.2 *The Cyclic-flip protocol is an m-bounded dynamic ∞-counter protocol.*

We need to prove that in every run in which at most m `increment` operations are performed, each call to a look or increment function terminates and produces the right result. This will follow from the following lemmas, which consider such a run. For the rest of this subsection we assume runs of the Cyclic-flip protocol in which there are at most m `increment` operations.

Lemma 5.3 *Any look or increment operation always terminates.*

```
/* $r_1, ..., r_m$ are the shared counter registers */
/* $A$ and $B$ are local arrays of size $m$ */

function **look**;
begin
      return $cyc(double\text{-}scan)$;
end;

procedure **increment**;
begin
      $flag := false$;
      repeat
            $A := double\text{-}scan$;
            $B := next(A)$;
            $j := 0$;
            repeat $j := j + 1$ until $A[j] \neq B[j]$;
            lock($r_j$)
                  if $A[j] = r_j$ then $\{r_j := B[j];\ flag := true\}$;
            unlock
      until $flag$;
end;

function **double-scan**: returns binary word of length $m$;
begin
      for $i := 1$ to $m$ do $A[i] := r_i$;
      repeat
            B:=A; /* array assignment */
            for $i := 1$ to $m$ do $A[i] := r_i$;
      until A=B; /* array test */
      return $(A)$;
end;

functions **cyc** and **next** are as defined earlier.
```

Figure 4: The Cyclic-flip protocol

Proof: First we prove that any double-scan operation terminates after reading each of the registers at most $m+2$ times. In the double-scan function, a process scans the registers from right to left, repeatedly until two successive scans return the same value. Whenever two successive scans return different values, some `increment` operation must have happened in the meantime. Since there are at most m `increment` operations, at most $m+1$ scans can return distinct values. Thus, the total number of scans is at most $m+2$.

Since a `look` operation includes a single call to the double-scan function, it follows from the above observation that any `look` operation terminates after reading each of the registers at most $m+2$ times.

Next we show that every `increment` operation terminates after at most $2m$ scans and m *rmw* operations. The worst possible run is where between any two successive changes of registers values during the run, at most two scans are performed. Since at most $m-1$ register changes, by other `increment` operations, may occur during this execution, at most $2m$ scan operations may be performed. After each such scan one *rmw* operation is performed, thus a total of at most m *rmw* operations are required. ▮

We say that a double-scan operation returns a *correct snapshot* of the counter registers if there is a point during the double-scan operation where the values of the registers are the same as the values returned.

Lemma 5.4 *Let x be a run with at most m increment operations, in which all the double-scan operations return a correct snapshot. Then*

1. *Whenever an rmw step in x changes the value of a register, the contents of the counter is changed from v to $next(v)$ (for some vector v).*
2. *Each register is changed at most once.*

Proof: 2 follows from 1 and Lemma 5.1. We prove 1 by induction on the number of *rmw* steps in x that actually change the value of a registers.

Induction Base: Consider the first *rmw* step that changes a value of a register during x, and let v^0 be the value returned by the double-scan function of the `increment` operation that executed this *rmw* step. By the assumption, v^0 is a correct snapshot of the counter. Since we consider the first *rmw* step that changed the value of a register, v^0 was also the value of the counter immediately before the *rmw* occured. The claim now follows by the definition of the `increment` procedure.

Induction Step: Assume that the lemma holds for the first k changes of registers values ($k < m$), and show that it holds also for the $k+1$-st change. Let v^i denote the value of the counter after the i'th change of the register value. Thus, we have that for $i \leq k$, $v^i = next(v^{i-1})$, and we must show that $v^{k+1} = next(v^k)$. For this, it suffices to show that the value returned by the double-scan of the `increment` operation of the $k+1$-st *rmw* step that changed the value of a register was v^k. By the assumption, the value returned by that double-scan is a correct snapshot of the counter, and hence is v^i for some $0 \leq i \leq k$. Now, if it is not v^k, then the value of the register on which the *rmw* step was performed was changed at least once after the double-scan operation and before the *rmw* step. By the induction hypothesis and Lemma 5.1, it could not be changed more than once. Hence, it was changed exactly once. This means that the *rmw* did not change the value of that register - a contradiction. hence the value returned by the double-scan operation is v^k, and by the definition of the `increment` procedure, $v^{k+1} = next(v^k)$. ▮

Lemma 5.5 *The double-scan operation always returns a correct snapshot of the counter registers.*

Proof: The proof by induction on the number of completed double-scan operations in a given run.

For the base of the induction we show that the first completed double-scan operation in any given run returns a correct snapshot of the counter registers. In order to change the value of the counter a process must first complete a double-scan operation. Hence during the first double-scan operation in any run, no register is changed and hence it must return a snapshot which is the actual initial contents of the registers.

We now assume that the lemma holds for any run in which at most k double-scan operations are completed, for some $k < m$, and we show that this is also the case for a run in which $k + 1$ double-scan operations are completed.

We observe that if a process reads twice the same value from some register then the value of the register was not changed between this two reads or it was changed an even number of times. Moreover, it means that if a process scans the registers twice, and read twice the same value from some register, then either:

1. The value of no register was changed during the first of the two scans (and hence the values returned by this double-scan operation is a correct snapshot of the counter); or
2. The value of some register was changed more than once during the two scans.

It follows from Lemma 5.4 that during the $k + 1$ double-scan operation each of the register was changed at most once. Hence, only the first case above is possible. ▮

We now complete the proof of Theorem 5.1, by showing that the cyclic-flip protocol satisfies the theorem. It is sufficient to prove that in every run of the protocol with at most m increment operations, every `look` operation returns a correct value of the counter. By Lemma 5.5 it returns a correct snapshot of the counter value. Let this snapshot be a vector v. Then by lemmas 5.4 and 5.2 $val(v) = cyc(v)$ is a correct value of the counter. ▮

Also, we point out that to implement the Cyclic-flip protocol it is enough to assume *test-and-set* atomicity rather than read-modify-write atomicity. (In the test-and-set operation the read together with the write is atomic but the the value written cannot depend on the value read.)

5.2 Dynamic n-counter Protocol

In this subsection we combine the Gray code protocol and the Cyclic-flip protocol to obtain an efficient (unbounded) dynamic n-counter protocol.

Theorem 5.3 *For any n, there is a dynamic n-counter protocol that uses $\alpha(\log m + 1)$ shared registers, where α is the smallest power of 2 that is not smaller than n.*

Let α be the smallest power of 2 that is not smaller than n. Each process starts the execution by assigning itself a unique identifier. This is done by using the Cyclic-flip counter protocol described in the previous subsection. We use α registers to implement such a counter. Each process as it wakes up increments this counter by 1, and assigns itself the new value.

Apart from this counter there are n other counters numbered 0 through $n-1$, each consisting of $\log m$ registers. Once a process assigns itself an identifier, say i, it considers counter i as its own local counter that only it can increment and everybody else can read. A process executes an `increment` operation by incrementing its local counter using the Gray code protocol. A process executes a `look` operation using the 4-way scan operation described in the Gray code protocol, on each of the n local counters, and then sums up the results.

The correctness of this protocol follows from the correctness proofs for the Gray code and Cyclic flip protocols.

6 Lower Bounds

In this section we prove lower bounds on the number of registers needed to implement concurrent counters (mod m), for $m > 2$. All the results that we prove in this section hold even if it is assumed that `look` operations are atomic. Recall that an *ℓ-bounded* dynamic n-counter is a dynamic n-counter which is guaranteed to be correct only in runs in which no more than ℓ `increment` operations are performed. Note that $\min\{\log m, \log \ell\}$ is a trivial lower bound on the number of registers needed to implement such a counter. The main result of this section is the following:

Theorem 6.1 *Let $k = \min(\frac{m+1}{2}, \frac{n+1}{2}, \sqrt{\frac{\ell+1}{3}})$. Any ℓ-bounded dynamic n-counter protocol must use at least k registers. This bound holds even when the processes are distinct and there is only one possible initial state.*

Proof: Assume to the contrary that there exists an ℓ-bounded dynamic n-counter protocol that uses $k < \min(\frac{m+1}{2}, \frac{n+1}{2}, \sqrt{\frac{\ell+1}{3}})$ registers. This implies that $n \geq 2k$, $m \geq 2k$, and $\ell \geq 3k^2$. We show how this assumption leads to a contradiction.

To make the result stronger we assume that there is only one possible initial state. To simplify the presentation we assume that in this single initial state the value of the counter is zero.

For $b \in \{0, 1\}$, we say that process p is *b-loaded* for register r in a run x if p's next step is a *rmw* step on r, and if the value of r is b then p will change it (to $\neg b$). Notice that by axiom *RMW1*, if p is b-loaded for r at x then p is also b-load for r at any run that is indistinguishable to p from x, provided that the value of r at that run is b. We now construct a run ρ as follows:

> Repeat the following procedure at most $2k$ times: Starting with $i = 1$, let process p_i repeatedly increment the counter until one of the following two situations

happens: (1) p_i becomes the first b-loaded process for some register r, (and in this case we say that p is *suspended* on r); or (2) p_i completes an `increment` operation and the total number of `increment` operations that have begun so far is ℓ. If (1) happens, and we still have not suspended $2k$ processes, then we activate process p_{i+1} according to the above procedure (and increment i by one). The construction terminates when either (2) happens or $2k$ processes have been suspended already.

We note that at most two processes may become *suspended* on each register (one may become 0-loaded and the other may become 1-loaded), and since $n \geq 2k$ we have always a process to activate when needed.

An `increment` operation that was completed in the run ρ is *reversible* if for any register that was changed during this operation there were already two processes suspended on it, and is *irreversible* otherwise. It is not difficult to see that at most k `increment` operations in ρ are *irreversible*. This follows from the fact that every register can be changed at most once during the run ρ while there is only one process suspended on it.

Next we show that if $2k$ processes are suspended at ρ then we are done. If this is the case, then for any of the k registers there is a 0-loaded process and a 1-loaded process suspended on it. This means that by activating some of these processes we have full control over the value of the counter.

Let ρ be such a run, and let $I = \mathit{end}(\rho) \oplus k$ (where, as before, $\oplus$ denotes addition modulo m, and $\mathit{end}(\rho)$ is the number of increments that have been completed in ρ). W.l.o.g. assume that when the value of the counter is $I \oplus (k+1)$, the values of the counter registers $r_k, ..., r_1$ are $v_k, ..., v_1$, respectively, and that in the run ρ, for each register r_i process p_i is $(\neg v_i)$-loaded for r_i in ρ. Activate in some order all suspended processes except processes $p_1, ..., p_k$, and let them terminate. Now we still have the k processes suspended and we can activate a subset of them (depending on the current values of the counter registers) and get the counter equal $I \oplus (k+1)$. That is, we can extend ρ to a run ρ' where $\mathit{count}(\rho') = I \oplus (k+1)$ and $\mathit{end}(\rho') = I$. Since at most k processes are suspended in ρ', $\mathit{count}(\rho')$ must lie in the cyclic interval $[I, I \oplus k]$. Since it is assumed that $k \leq m/2$, $I \oplus (k+1)$ does not lie in the cyclic interval $[I, I \oplus k]$, a contradiction.

So, let us assume from now on that at most $2k-1$ processes are suspended at ρ. This implies that the total number of `increment` operations that have begun in ρ is exactly ℓ. Hence, the total number of (completed) reversible `increment` operations is at least $\ell - (2k-1) - k$. We call each maximal sequence of consecutive events of ρ which contains only reversible `increment` operation, a *segment*. (I.e., a segment does not contain any irreversible `increment` operations and any operations by suspended processes.) We notice that two consecutive segments are separated by one or more irreversible `increment` operations or operations by (eventually) suspended processes. Since there are at most k irreversible `increment` operations, and at most $2k-1$ processes are suspended at ρ, there are at most $3k$ segments. Since there are at least $\ell - 3k + 1$ reversible `increment` operations, $\ell \geq 3k^2$ and k is an integer, there must be at least one segment which includes (at least),

$$\left\lceil \frac{\ell - 3k + 1}{3k} \right\rceil \geq \left\lceil \frac{3k^2 - 3k + 1}{3k} \right\rceil = \left\lceil k - 1 + \frac{1}{3k} \right\rceil = k$$

reversible `increment` operations.

Let x and y be two prefixes of ρ ($x \leq y$) such that $(y - x)$ includes exactly k reversible `increment` operations, does not include any irreversible `increment` operations, all `increment` operations in x and y, except those of processes that are suspended in x, has been completed, and no new processes are suspended in $(y - x)$. Such two prefixes must exist by the argument in the previous paragraph.

Since $(y - x)$ includes only reversible `increment` operations, a register r is changed during $(y - x)$ only if two processes are suspended on r at x.

By definition of reversible `increment` operations we can now activate some of the processes that are suspended at y to get an extension z of y such that the values of all the counter registers are the same in x and z. This implies that $count(x) = count(z)$. As already mentioned, a value of a register r was changed in $(y - x)$ only if two processors were suspended on r in x. Since there are at most $k - 1$ such registers, at most $k - 1$ processes are needed to be activated in $(z - y)$. Also, the runs x and z are indistinguishable to every process suspended in x that is not involved in some event in $(z - y)$. Finally, we extend x and z to runs x' and z' respectively, in exactly the same way, by letting all the suspended process in x that are not involved in some event in $(z - y)$ complete their `increment` operations.

Next we show that at most $k - 1$ processes are in the middle of an `increment` operation in z' (and also in x'). This follows from the fact that in the only processes that did not complete their `increment` operation in z' (and in x') are those which were activated in $(z - y)$, and as said above there are at most $k - 1$ such processes.

From the construction, using the *RMW* axioms it follows that the values of all the counter registers are the same in x' and z' and thus $count(x') = count(z')$. On the other hand, let $end(x')$ be the number of increments that have been completed in x'. Since at most $k - 1$ processes are in the middle of an `increment` operation in x', $count(x')$ must lie in the cyclic interval $[end(x'), end(x') \oplus (k - 1)]$. Also, since $end(z') \geq end(x') + k$ and since at most $k - 1$ processes are in the middle of an `increment` operation in z', $count(z')$ must lie in the cyclic interval $[end(x') \oplus k, end(x') \oplus (k + k - 1)]$. Since, $k \leq m/2$, it must be that $count(z') \neq count(x')$, a contradiction. ▮

By slightly modifying the proof of Theorem 6.1, we can prove the following: Let $k = \min(\frac{m+1}{3}, \frac{n+1}{2}, \sqrt{\ell + 4.25} - 1.5)$. Any ℓ-bounded dynamic n-counter protocol must use at least k registers. (Again, this bound holds even when the processes are distinct and there is only one possible initial state.)

It follows from Theorem 6.1 that any dynamic n-counter protocol must use at least $\min(\frac{m+1}{2}, \frac{n+1}{2})$ bits. Next, by making various restrictions on the way processes may increment the counter, we tighten the lower bound. Recall that a protocol is a *1-flip* protocol if a process may change the value of only one register during a single `increment` operation.

Theorem 6.2 *Let $k = \min(l, n)$. Any 1-flip ℓ-bounded static n-counter protocol must use at least $k - 1$ registers.*

Proof: Assume to the contrary that $k - 2$ registers are sufficient. Consider a run ρ where $k - 1$ different processes increment the counter in a sequential manner, one time each, starting from some arbitrary initial state. Since there are only $k - 2$ registers the value of at least one register, say r, is changed at least two times. Let p_1 be the process that was the first to change r and let p_2 be the process that was the second to change r. Let ρ_2 be a prefix of ρ where p_2 just completed its `increment` operation, and let ρ_1 be a prefix of ρ_2 where p_2 is just about to start its `increment` operation.

Next we construct the run ρ_3 as follows. Let p_3 be a process that did not participate in ρ_2. We construct ρ_3 by first activating the processes exactly as in ρ_2 until the point where p_1 is about to change r for the first time. Then we activate p_3 until it is also about to change r. This will happen since processes p_1 and p_3 are identical. Then we suspend p_3 and let the run continue as in ρ_2. Finally, after p_2 change the value of r for the second time (and completed its `increment` operation) we activate p_3 and let it changes r for the third time and complete it `increment` operation. Notice that between the point where p_3 was suspended and the point when it was activated again the value of r is changed exactly two times and hence p_3 will not notice that r has been changed and will change r when it is activated again.

Since in ρ_1 the register r was changed once and in ρ_3 it was changed three times the values of all registers in these two runs are the same and hence $count(\rho_1) = count(\rho_3)$. However, the number of `increment` operation in ρ_3 is greater by exactly two than in ρ_1, and since it is assumed that $m > 2$, we reach a contradiction. ∎

It follows from Theorem 6.2 that there does not exist 1-flip static ∞-counter protocol when using only binary registers. Theorem 6.2 can easily be generalized to show that when $m > v$, any 1-flip ℓ-bounded static n-counter protocol must use at least $(k-1)/(v-1)$ v-valued registers, where $k = \min(l, n)$.

The implementations of the `increment` operation in all the protocols we present in this paper, except the protocol in Section 5.2, have the property of being independent of the specific state of the process that executes them. In every single `increment` operation the final contents of the counter is determined only by its initial contents. We call such a counter protocol a *memory-less* counter protocol. For such protocols, we have a stronger version of Theorem 6.2.

Theorem 6.3 *Any 1-flip memory-less ℓ-bounded static 2-counter protocol must use at least $\ell - 1$ registers.*

The proof of Theorem 6.3 is similar to that of Theorem 6.2 except that in the construction of the run ρ in the previous proof, instead of activating $k-1$ different processes, we activate only one process and let it increment the counter $k-1$ times.

We point out that Theorem 6.3 does not contradict Theorem 5.3, since the protocol described in Section 5.2 is not a memory-less or a 1-flip protocol. As in Theorem 6.2, it follows from Theorem 6.3 that there does not exist a 1-flip memory-less static 2-counter protocol when using only binary registers.

7 Discussion

In this paper we studied a new basic problem – the concurrent counter problem – in a model where no assumption is made about the initial state of the shared memory. We presented some protocols for solving the problem, and proved some lower bounds on its space complexity.

Let the time complexity be the total number of accesses to the shared memory in order to complete a `look` or `increment` operation. The time complexity of both the `look` and `increment` operations in the Positional protocol is $\log m$. In the Gray-code protocol the time complexity of the `look` operation is $4 \log m$, and the time complexity of the `increment` operation is 1 apart from the first `increment` which takes $\log m + 1$. As for the Cyclic-flip protocol, in the absence of contention, the time complexity of the `look` operation is $2m$, and the time complexity of the `increment` operation is $2m + 1$. When there is contention the complexity can be in the worst case $m^2 + 2m$ for the `look` operation and $2m^2 + m$ for the `increment` operation. As for the protocol discussed in subsection 5.2, the time complexity of the `look` operation is $n \log m$, while the time complexity of `increment` operation is 1, apart from the first `increment` operation which may take $2m^2 + m + 1$.

There are still many interesting open questions related to concurrent counters. Some of these problems are listed below.

The lower bound in Theorem 6.1 is not tight. Improving this lower bound (or the corresponding upper bound) may also help in improving the related time bound, and may have implications on the bounds in [FMRT90].

Generalize or modify counters to objects that support a wider variety of operations. For example, a natural generalization whose implementation raises non-trivial problems is obtained by extending the counter definition to allow a *decrement* operation, which decreases the value of the counter by one. Another operation is to *reset* the counter to some default value.

Acknowledgment: We would like to thank Hagit Brit for helpful comments, and the two anonymous referees for a thorough review of the manuscript.

References

[AA92] E. Aharonson and H. Attiya. Counting networks with arbitrary fan-out. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 104–113, January 1992.

[AG91] J. H. Anderson and B. Grošelj. Beyond atomic registers: Bounded wait-free implementations of nontrivial objects. In *Fifth International Workshop on Distributed Algorithms*, October 1991.

[AH90a] J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–460, September 1990.

[AH90b] J. Aspnes and M. Herlihy. Wait-free data structures in the asynchronous PRAM model. In *Proceedings of the second Annual ACM symposium on parallel architectures and algorithms*, pages 340–349, July 1990.

[AHS91] J. Aspnes, M. Herlihy, and N. Shavit. Counting networks and multi-processor coordination. In *Proc. 23rd ACM Symp. on Theory of Computing*, pages 348–358, May 1991.

[AVY94] W. Aiello, R. Venkatesan, and M. Yung. Coins, weights and contention in balancing networks. In *Proc. 13th ACM Symp. on Principles of Distributed Computing*, pages, 193–205, August 1994.

[BIS95] A. Ben-Dor, A. Israeli and A. Shirazy. Dynamic Counting. In *Proc. 3rd Israel Symposium on the theory of computing and systems*, pages 111–120, January 1995.

[BM94a] H. Brit and S. Moran, Wait-freedom vs. Bounded Wait-freedom in Public Data Structures, In *Proc. 13th ACM Symp. on Principles of Distributed Computing*, August 1994.

[BM94b] C. Busch and M. Mavronicolas. A combinatorial treatment of balancing networks. In *Proc. 13th ACM Symp. on Principles of Distributed Computing*, August 1994.

[BMT95] H. Brit, S. Moran, and G. Taubenfeld. Public data structures: counters as a special case. In *Proc. 3rd Israel Symposium on the theory of computing and systems*, January 1995.

[Dij74] E. W. Dijkstra. Self-stablizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.

[Fis83] M. J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In M. Karpinsky, editor, *Foundations of Computation Theory*, pages 127–140. Lecture Notes in Computer Science, vol. 158, Springer-Verlag, 1983.

[FMRT90] M.J. Fischer, S. Moran, S. Rudich, and G. Taubenfeld. The wakeup problem. In *Proc. 22st ACM Symp. on Theory of Computing*, pages 106–116, May 1990. To appear in *SIAM Journal on Computing.*

[FMT93] M.J. Fischer, S. Moran, and G. Taubenfeld. Space-efficient asynchronous consensus without shared memory initialization. *Information Processing Letters*, 45:101–105, 1993.

[Gar72] M. Gardner. Mathematical games: The curious properties of the Gray code and how it can be used to solve puzzles. *Scientific American*, pages 106–109, August 1972.

[Gil58] E. N. Gilbert. Gray codes and paths on the n-cube. *The bell system technical journal*, 37(9):815–826, May 1958.

[Gra53] F. Gray. Pulse code communication. U. S. Patent 2 632 058, March 1953.

[HBS92] M. Herlihy, Lim. B., and N. Shavit. Low contention load balancing on large-scale multiprocessors. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, July 1992.

[HSW91] M. Herlihy, N. Shavit, and O. Waarts. Low contention linearizable counting. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pages 526–535, October 1991.

[HW90] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, July 1990.

[KMZ84] E. Korach, S. Moran, and S. Zaks. Tight lower and upper bounds for some distributed algorithms for a complete network of processors. In *Proc. 3rd ACM Symp. on Principles of Distributed Computing*, pages 199–207, 1984.

[KP92] M. Klugerman and C. Plaxton. Small-depth counting networks. In *Proc. 24rd ACM Symp. on Theory of Computing*, pages 417–428, October 1992.

[Koh70] Z. Kohavi. *Switching and Finite Automata Theory.* McGrow-Hill Publication, 1970.

[Lam77] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20:806–811, 1977.

[Lam86] L. Lamport. On interprocess communication, parts I and II. *Distributed Computing*, 1(2):77–101, 1986.

[Lam90] L. Lamport. Concurrent reading and writing of clocks. *ACM Trans. on Computer Systems*, 8(4):305–310, 1990.

[MT93] S. Moran and G. Taubenfeld. A lower bound on wait-free counting. In *Proc. 12th ACM Symp. on Principles of Distributed Computing*, pages 251–260, August 1993.

[MTY92] S. Moran, G. Taubenfeld, and I. Yadin. Concurrent counting. In *Proc. 11th ACM Symp. on Principles of Distributed Computing*, pages 59–70, August 1992.

[Pet82] G. L. Peterson. An $O(n \log n)$ unidirectional algorithm for the circular extrema problem. *ACM Trans. on Programming Languages and Systems*, 4(4):758–762, 1982.

[SZ94] N. Shavit, and A. Zemach. Diffracting trees, In *Annual Symp. on Parallel Algorithms and Architectures*, June 1994.