Computing in Totally Anonymous Asynchronous Shared Memory Systems*

Hagit Attiya† Alla Gorbach‡ Shlomo Moran§

Department of Computer Science
The Technion, Haifa 32000, Israel

July 13, 1999

Abstract

In the *totally anonymous* shared memory model of asynchronous distributed computing, processes have no id's and run identical programs. Moreover, processes have identical interface to the shared memory, and in particular, there are no *single-writer* registers. This paper assumes that processes do not fail, and the shared memory consists only of read/write registers, which are initialized to some default value.

A complete characterization of the functions and relations that can be computed in this model is presented. The consensus task is an important relation which can be computed. Unlike functions, which can be computed with two registers, the consensus protocol uses a linear number of shared registers and rounds.

The paper proves logarithmic lower bounds on the number of registers and rounds needed for solving consensus in this model, indicating the difficulty of computing relations in this model.

---

*An extended abstract of this paper appeared in proceedings of the *12th International Symposium on Distributed Computing*, September 1998, (S. Kutten, ed.), pp. 49–61, Lecture Notes in Computer Science #1499, Springer-Verlag.

†`hagit@cs.technion.ac.il`. Work supported by the fund for the promotion of research in the Technion.

‡`allag@msil.sps.mot.com`.

§`moran@cs.technion.ac.il`. Research supported by the Bernard Elkin Chair for Computer Science. Part of the work was done while this author was at the University of Arizona, supported by US-Israel BSF grant 95-00238.

1 Introduction

In this work, we study the *totally anonymous shared memory* model of asynchronous distributed computing. Like in previous works that studied computations by identical processes, e.g., [2, 3], we assume that processes have no id's and run identical programs. We also assume that the means by which processes access the shared memory are identical to all processes. This implies, for example, that a process cannot have a private register to which only this process may write and all other processes can read, as is usually assumed.

This model is related to an environment where an unknown set of processes execute a common task using a "public" server, in which the communication media cannot distinguish between different processes. This describes an environment where the connection of a process to the shared server can come up dynamically.

We are mainly interested in the computation power of the totally anonymous model, namely, which distributed tasks [16] can be computed by it. We assume that processes do not fail, and that the shared memory consists only of read/write shared registers, which are initialized to some default value. We mostly assume that the number of processes that execute a given task is unknown. We restrict our attention to consensus-like tasks, in which each process has a private input, and processes have to agree on the same output. Such a task defines a *relation* between input vectors and possible outputs. A relation is a *function* if the corresponding output is uniquely determined by the processes inputs (e.g., the AND function).

An important example of a relation is the *consensus* task [12, 17], where the output should be the input of some process and all processes should agree on it. Consensus is a fundamental problem in distributed computing: Solving consensus means reaching agreement between remote processes, which is an essential part of many distributed algorithms. In asynchronous systems, consensus cannot be solved in the presence of failures [8].

We present a full characterization of the functions that can be computed in this model, and show that if a function is computable, then it can be computed with only two shared registers. The characterization puts prohibitive limitations on the input domain of computable functions. We show that many natural functions, e.g., Boolean AND, cannot be computed in this model. On the other hand, we show that some non-trivial functions are computable in this model.

We introduce a (fault-free) consensus protocol in which both the number of shared registers and the number of rounds is linear in the number of processes. Then, we give a complete characterization of the relations computable in the model. The consensus protocol is used to prove the sufficiency condition of the characterization.

We present lower bounds on the complexity of solutions for computable tasks. We prove an $\Omega(\log N)$ lower bound on the number of registers needed for solving consensus. The proof combines two techniques: The bivalence argument of Fischer, Lynch and Paterson [8] and the covering technique introduced by Burns and Lynch [5]. We also prove an $\Omega(\log N)$ lower bound on the round complexity needed for solving consensus. This proof also uses the bivalence argument [8]. Since consensus is one of the computable relations in our model, the lower bounds on the complexity of solving consensus imply lower bounds on every generic protocol for computing relations, indicating the difficulty of computing relations in our model.

Jayanti and Toueg [11] studied two anonymous read/write shared memory models: In the first model, that is only partially anonymous, the processes have no names but the shared registers are single writer (each process has its "own" register). The second model is similar to ours (it is totally anonymous), but the shared memory is not initialized to a known state. The definition of the models immediately imply that our model is at least as powerful as the totally anonymous model of [11]. In the first model, the *wakeup* task [9] and the consensus task [12, 17] can be solved, but the leader election task cannot be solved; in the second model, all these tasks, and in particular, consensus, cannot be solved. In our model, consensus can be computed but wakeup cannot (the impossibility result of [11] is valid for our model as well). Therefore, our model is less powerful than the first, partially anonymous model of [11] and it is more powerful than their second, totally anonymous model.

Burns and Lynch [5] proved that every protocol for mutual exclusion among N processes, using read/write shared registers, requires N shared registers. (Mutual exclusion cannot be solved in the totally anonymous model.) Our proof of the lower bound on the number of registers for consensus uses a technique of "covering" registers, which is similar to the technique used by Burns and Lynch. Different variants of this technique appear in the "hiding lemma" used by Moran and Taubenfeld [15] to prove impossibility of wait-free counting, and in the "reconstruction lemma" used by Brit and Moran [4] to prove equivalence between wait-freedom and bounded wait-freedom in periodic public data structures.

Styer and Peterson [18] studied mutual exclusion and leader election in read/write shared memory model, where processes are symmetric—they have identifiers which can only be compared. Again, these tasks cannot be solved in the anonymous model. Styer and Peterson proved that N registers are necessary for deadlock-free mutual exclusion and they must be written and that $2N-1$ registers are necessary for lockout-free mutual exclusion in the memoryless system. (This extends the lower bound of Burns and Lynch [5] and uses their covering technique.) Styer and Peterson also proved that $\lceil \log N \rceil + 1$ registers are necessary for leader election between N processes. This result implies that $\lceil \log N \rceil + 1$ registers are required for *general* (non-binary) consensus in our model. Our lower bound holds for binary consensus, and hence it does not follow from the lower bound of Styer and Peterson [18].

Fischer, Lynch and Paterson [8] proved the impossibility of asynchronous consensus in the presence of one faulty process. This proof introduces an important tool for impossibility and lower bound proofs—the existence of bivalent states—which was used extensively afterwards, e.g., [6, 10, 13, 14], mainly in the context of impossibility results in fault tolerant computations. We use bivalence arguments in our lower bounds, demonstrating an interesting connection between models with distinct, possibly faulty processes, and models with totally anonymous, nonfaulty processes.

Fich, Herlihy and Shavit [7] studied the space complexity of solving randomized consensus. They proved that if all processes are identical and only read/write operations are used, then every solution for randomized wait-free consensus uses $\Omega(\sqrt{N})$ registers. They extended this result for non-identical processes and history-less objects. This cannot be done for our lower bound for consensus, since if processes have identifiers, consensus can be solved using a constant number of registers in a manner similar to the protocol for computing functions given in Section 3. These lower bounds also use the covering technique [5].

2 Preliminaries

2.1 The model

We consider an asynchronous system of $N \geq 2$ anonymous nonfaulty processes which communicate via read/write shared registers. Each process can read from or write to every shared register. For ease of exposition, we refer to the processes by the unique names $p_1, p_2, \ldots, p_N$, but the names are not available to the processes themselves.

Each process p_i has local *input* and *output* registers and an unbounded amount of *internal storage*. The input and output registers together with the internal storage comprise the *internal state* of p_i. *Initial states* prescribe starting values to the input register and the internal storage, and the *empty* value to the output register. A process is in a *termination state* if its output register is not empty.

Process p_i acts according to a *transition function*, which determines for each internal state of a p_i what is the next action of p_i and what is the new internal state of p_i. The next action of p_i includes whether this is a read or a write, to which register and what is the value written (if this is a write). The new internal state of p_i depends on the previous state of p_i and the result of the action (if this is a read).

A *protocol* is a specification of an initial state and a transition function; we assume these are the same for all processes.

We assume that the protocol is *uniform*, that is, N, the total number of processes in the system, is unknown, unless stated otherwise.

A *configuration* consists of the internal state of each process, together with the values of the shared registers. An *initial configuration* of a protocol $\mathcal{P}$ requires every process to be in the initial state prescribed by $\mathcal{P}$ and the shared registers to contain default values. Note that the shared memory is initialized to some known state.

A *step* takes one configuration to a successive configuration and involves a primitive step of a single process. A primitive step of process p in the model is one of the following types:

1. $read_p(r)$: p reads the value of the register r.

2. $write_p(r, v)$: p writes the value v to the register r.

3. nop_p: p does nothing.

The read and write steps are performed by process p only if it is not in a termination state, while the nop step is performed only if p is in a termination state.

All following definitions are with respect to a specific protocol $\mathcal{P}$.

If p is a process and C is a configuration, then $p(C)$ denotes the configuration obtained after p takes a step in C according to the transition function of $\mathcal{P}$.

A *schedule* is a sequence of process names. If $\sigma = p_{i_1}, p_{i_2}, \ldots, p_{i_n}$ is a finite schedule and C is a configuration, then $\sigma(C)$ denotes $p_{i_n}(p_{i_{n-1}}(\ldots p_{i_1}(C)\ldots))$.

An *execution* of $\mathcal{P}$ is a (finite or infinite) sequence of the form

$$C_0, p_{i_1}, C_1, p_{i_2}, C_2, p_{i_3}, \ldots,$$

where for every k, C_k is a configuration, p_{i_k} is a process, and C_k is $p_{i_k}(C_{k-1})$. An execution describes a sequence of successive configurations that are generated when processes take steps.

An initial configuration and a schedule uniquely determine an execution, and vice versa. Given an execution $\alpha = C_0, p_{i_1}, C_1, p_{i_2}, C_2, \ldots$, $\mathit{sched}(\alpha) = p_{i_1}, p_{i_2}, \ldots$ is the schedule that *corresponds* to α. Given a schedule $\sigma = p_{i_1}, p_{i_2}, \ldots$ and a configuration C, the execution that results from applying the schedule σ from the configuration C, denoted (C, σ), is the execution

$$C, p_{i_1}, p_{i_1}(C), p_{i_2}, p_{i_2}(p_{i_1}(C)), \ldots.$$

An infinite schedule σ is *fair* if each process appears infinitely many times in σ. An infinite execution α is *fair* if $\mathit{sched}(\alpha)$ is fair.

If σ_1 is a finite schedule and σ_2 is a schedule, then $\sigma_1 \cdot \sigma_2$ is the schedule obtained by concatenating σ_1 and σ_2. If $\sigma = \sigma_1 \cdot \sigma_2$, we denote σ_2 by $\sigma - \sigma_1$.

If α is a finite execution and σ is a schedule, then (α, σ) denotes the extension of α with the schedule σ: if C_0 is the first configuration in α then (α, σ) is $(C_0, \mathit{sched}(\alpha) \cdot \sigma)$. If α is a finite execution, then $C_{end}(\alpha)$ is the last configuration in α.

2.2 Computing functions and relations

In its most general setting, a distributed *task* T is a relation, consisting of a set of pairs $(\overline{X}, T[\overline{X}])$, where $\overline{X}$ is an input vector and $T[\overline{X}]$ is a non-empty set of output vectors that are allowable for $\overline{X}$ [16]. A vector $\overline{X}$ is a *legal input vector* for a task T if there is a pair $(\overline{X}, S)$ in T. The set of legal input vectors for T is the *domain* of T and denoted by $\mathcal{D}_T$. In this work we restrict our attention to tasks in which all processes agree on the same output; in this case, we represent each output vector in $T[\overline{X}]$ as a scalar, with the obvious meaning.

A task T is a *function* if the corresponding relation is a function—for every legal input vector, $\overline{X}$, $T[\overline{X}]$ is a singleton. We use the letter R to denote a task which is a relation, but not necessarily a function, and the letter f to denote a task which is a function.

A typical example of a relation is *K-valued consensus*. The inputs for this relation are vectors of integers from the range $\{0, \ldots, K-1\}$. All the values that appear in an input vector $\overline{X}$ are allowed outputs for $\overline{X}$.

A protocol $\mathcal{P}$ *computes* (or *solves*) a relation R if for every vector $\overline{X} = (x_1, x_2, \ldots, x_N) \in \mathcal{D}_R$ and for every fair execution α such that p_i has an input x_i in α:

1. every process eventually writes a *decision* $d \in R[\overline{X}]$ into its local output registers in α; and

2. the decisions of all processes are the same.

If process p writes a decision d in execution α, we say that p *decides* on d in α.

2.3 Basic properties of the model

A basic property of the totally anonymous model is that a single process can be replaced with a group of processes in which every process has the same input, and vice versa. In this section, we define notions of *multiplication* and *equivalence*, capturing this property. The impossibility results and lower bounds presented later rely on these notions.

Given a finite execution α, the *view* of a process p after α, denoted $\alpha|p$, is the vector containing the internal state of p and the values of the shared registers in $C_{end}(\alpha)$. Notice that if $\alpha|p = \beta|q$ and p decides on v in α, then q decides on v in β as well.

Using the notion of a view, we define when one execution hides another one. Informally, an execution α hides an execution β if every process that appears in α cannot know if the execution is α or β. For example, if α is an execution of a single process p with input 0 and β is an execution of two processes—p with input 0 and q with input 1, where every write of q is immediately overwritten by p, then α hides β. As another example, consider an execution α' of two processes p_1 and p_2—both with input 0, such that p_1 and p_2 take their steps together (one step of p_1 followed by one step of p_2); α' also hides β.

The formal definition is as follows. A finite execution α of processes $p_1, \ldots, p_k$ *hides* a finite execution β of processes $q_1, \ldots, q_n$, denoted $\alpha \preceq \beta$, if for every p_i there is q_j such that $\alpha|p_i = \beta|q_j$. We also say that β is *hidden* by α.

If $\alpha \preceq \beta$ and $\beta \preceq \alpha$, i.e., they hide each other, then α and β are *equivalent*, denoted $\alpha \approx \beta$.

When α and β are infinite, $\alpha \approx \beta$ if there is an infinite sequence $\{\alpha_i\}_{i=0}^{\infty}$ of finite prefixes of α and an infinite sequence $\{\beta_i\}_{i=0}^{\infty}$ of finite prefixes of β such that $\alpha_i \approx \beta_i$, for every i.

Let σ be a schedule of processes $q_1, \ldots, q_n$, and let $\{P_1, \ldots, P_n\}$ be a set of groups of processes, where $P_i = \{p_i^1, p_i^2, \ldots, p_i^{l(i)}\}$, $i = 1, \ldots, n$. The *multiplication* of σ by $\{P_1, \ldots, P_n\}$ is the schedule obtained from σ by substituting every appearance of q_i in σ with the sequence $p_i^1, p_i^2, \ldots, p_i^{l(i)}$, for every i, $1 \le i \le n$.

Let α be an execution of processes $q_1, \ldots, q_n$ such that the input of q_i in α is x_i. β is the *multiplication* of α by $\{P_1, \ldots, P_n\}$ if process p_i^j has an input x_i in β for every i, $1 \leq i \leq n$, and j, $1 \leq j \leq l(i)$, and $sched(\beta)$ is the multiplication of $sched(\alpha)$ by $\{P_1, \ldots, P_n\}$.

The next lemma shows that an execution and its multiplication are very similar.

Lemma 2.1 *Let α be an execution of processes $q_1, \ldots, q_n$. If β is the multiplication of α by $\{P_1, \ldots, P_n\}$, then for every i, $1 \leq i \leq n$, and for any $p \in P_i$, $\alpha|q_i = \beta|p$, and the same set of registers is written in α and β.*

Proof: The proof is by induction on the length of α.

Basis: If the length of α is 0, then α and β are empty executions. Thus, the shared memory is in the initial state in $C_{end}(\alpha)$ and $C_{end}(\beta)$. Every process $p \in P_i$ is in the same state as q_i, because they have the same input. Hence, $\alpha|q_i = \beta|p$. No register is written in both executions.

Inductive step: Suppose the lemma holds for α; we prove it for α' which is the extension of α by a step of process q_k. Let β and β' be the multiplications of α and α' by $\{P_1, \ldots, P_n\}$, respectively. The shared memory is in the same state in $C_{end}(\alpha)$ and $C_{end}(\beta)$; for every i and every $p' \in P_i$, q_i and p' are in the same state in $C_{end}(\alpha)$ and $C_{end}(\beta)$.

For every $i \neq k$, the state of q_i in $C_{end}(\alpha')$ is the same as its state in $C_{end}(\alpha)$ and the state of $p' \in P_i$ in $C_{end}(\beta')$ is the same as its state in $C_{end}(\beta)$. Therefore, for every $i \neq k$ and any $p' \in P_i$, $\alpha'|q_i = \beta'|p'$.

Processes q_k and $p \in P_k$ are in the same state in $C_{end}(\alpha)$ and $C_{end}(\beta)$, respectively, and therefore, the next steps of q_k and p are the same. Since the memory state in $C_{end}(\alpha')$ is the same as in $C_{end}(\beta')$, the values read from the memory (if this is a read step) are the same; since q_k and $p \in P_k$ are in the same state, the value written to the memory (if this is a write step) are the same. Thus, q_k and p are in the same state in $C_{end}(\alpha')$ and $C_{end}(\beta')$, respectively; also, the memory state in $C_{end}(\alpha')$ is the same as in $C_{end}(\beta')$; therefore, $\alpha'|q_k = \beta'|p$.

By the inductive hypothesis, the sets of registers written in α and β are the same. In the extensions, either no register is written or the same register is written. Therefore, the sets of registers written in α' and β' are the same. ∎

Corollary 2.2 *If α is an execution and β is a multiplication of α, then $\alpha \approx \beta$.*

Since the model is anonymous, neither the order of values in the input vector nor the multiplicity of a value are significant, as formalized by the next definition and Lemma 2.3.

If $\overline{X} = (x_1, x_2, \ldots, x_N)$ is an input vector, then let $set(\overline{X})$ be the set of distinct values in $\overline{X}$. The *multiplicity* of $x_i \in set(\overline{X})$ in $\overline{X}$ is the number of appearances of x_i in $\overline{X}$; it is denoted $mult(x_i, \overline{X})$.

The next lemma shows that for an input vector $\overline{X}$, if α is an execution such that the input set in α is $set(\overline{X})$ and the multiplicity of each value is smaller than its multiplicity in $\overline{X}$, then processes have to decide on a value from $R[\overline{X}]$ in α.

Lemma 2.3 *Let $\mathcal{P}$ be a protocol that computes a relation R and let $\overline{X} \in \mathcal{D}_R$ be an N-entry input vector. Let $\overline{Z}$ be an n-entry vector, $n \leq N$, such that $set(\overline{Z}) = set(\overline{X})$ and for every $x \in set(\overline{X})$, $mult(x, \overline{Z}) \leq mult(x, \overline{X})$. In every fair execution of $\mathcal{P}$ with input $\overline{Z}$, processes decide on some $y \in R[\overline{X}]$.*

Proof: Denote $set(\overline{X}) = set(\overline{Z}) = \{x_1, \ldots, x_k\}$; let m_i be $mult(x_i, \overline{Z})$ and n_i be $mult(x_i, \overline{X})$.

Let α be a fair execution with input vector $\overline{Z}$; let $q_i^1, \ldots, q_i^{m_i}$ be the processes with input x_i in α. For every i, $1 \leq i \leq k$, let P_i^j be $\{p_i^j\}$ for every j, $1 \leq j < m_i$ and let $P_i^{m_i}$ be $\{p_i^{m_i}, \ldots, p_i^{n_i}\}$. Consider an execution β that is the multiplication of α by $\{P_1^1, \ldots, P_1^{m_1}, \ldots, P_k^1, \ldots, P_k^{m_k}\}$.

The input vector in β is $\overline{X}$; since α is fair for processes appearing in it, β is fair for processes appearing in it. Thus, in β, every process decides on $y \in R[\overline{X}]$. By Corollary 2.2, $\alpha \approx \beta$, and hence every process in α decides on $y \in R[\overline{X}]$. ■

3 Computing functions

3.1 A characterization

In this section, we prove the next characterization of the computable functions in our model.

Theorem 3.1 *A function f can be computed in the totally anonymous model if and only if $f[\overline{X}] = f[\overline{Y}]$, for every pair of input vectors $\overline{X}, \overline{Y} \in \mathcal{D}_f$ such that $set(\overline{X}) \subseteq set(\overline{Y})$.*

The multiplicity of values in the input vector does not affect the function's output, and thus, the characterization only refers to the input sets (and not the input vectors).

As an example, consider a function which outputs the minimum of N inputs with exactly k different values (for some *known* constant k). It is easy to check that for every pair of legal N-tuples $\overline{X}$ and $\overline{Y}$ such that $set(\overline{X}) \subseteq set(\overline{Y})$ it must be that $set(\overline{X}) = set(\overline{Y})$. Hence, $\min[\overline{X}] = \min[\overline{Y}]$.

We start with the necessary condition.

Lemma 3.2 *Given a function f, if there are input vectors $\overline{X}, \overline{Y} \in \mathcal{D}_f$ such that $set(\overline{X}) \subseteq set(\overline{Y})$ and $f[\overline{X}] \neq f[\overline{Y}]$, then f cannot be computed in the totally anonymous model.*

Proof: Suppose for contradiction that $\mathcal{P}$ is a protocol for computing f. Let $\overline{X}, \overline{Y}$ be input vectors such that $set(\overline{X}) \subseteq set(\overline{Y})$ and $f[\overline{X}] \neq f[\overline{Y}]$. Without loss of generality, let $set(\overline{X})$ be $\{x_1, \ldots, x_k\}$ and let $\overline{Y}$ be $(x_1, \ldots, x_k, x_{k+1}, \ldots, x_N)$.

Informally, we let k processes with inputs $x_1, x_2, \ldots, x_k$ run on their own, and eventually they decide on $f[\overline{X}]$. Then we let processes with inputs $x_{k+1}, \ldots, x_N$ run on their own; they

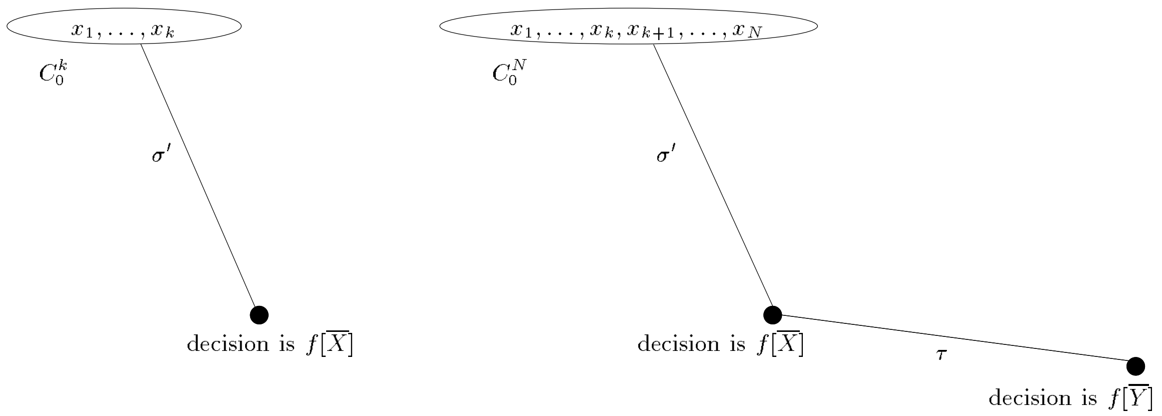

Figure 1: Illustration for the proof of Lemma 3.2.

have to decide on $f[\overline{Y}] \neq f[\overline{X}]$. Therefore, the decision of the first k processes is incorrect. The formal arguments follow (consider Figure 1).

Let C_0^k be an initial configuration with processes $q_1, \ldots, q_k$, where the input of q_i is x_i, and let σ be a fair schedule of $q_1, \ldots, q_k$. By Lemma 2.3, in a fair execution of $\mathcal{P}$ with input vector $(x_1, \ldots, x_k)$, processes decide on $f[\overline{X}]$. Thus, there is a finite prefix, σ', of σ such that $q_1, \ldots, q_k$ decide on $f[\overline{X}]$ in (C_0^k, σ').

Let C_0^N be an initial configuration with processes $q_1, \ldots, q_N$, such that the input of q_i in C_0^N is x_i. $(C_0^k, \sigma') \preceq (C_0^N, \sigma')$, and therefore, $q_1, \ldots, q_k$ decide on $f[\overline{X}]$ in (C_0^N, σ'). Let τ be a fair schedule of $q_1, \ldots, q_N$. Consider the execution $\gamma = (C_0^N, \sigma' \cdot \tau)$. The input vector in γ is $(x_1, \ldots, x_k, x_{k+1}, \ldots, x_N) = \overline{Y}$. Since τ is fair, γ is also fair and therefore, processes decide on $f[\overline{Y}]$ in γ. But $q_1, \ldots, q_k$ decide on $f[\overline{X}] \neq f[\overline{Y}]$ in γ, which is a contradiction. ■

To prove that the condition is sufficient, consider a function f satisfying the assumption of Theorem 3.1.

A set $Z = \{z_1, \ldots, z_k\}$ is *determining* for f if there is a value $val(Z)$ such that for every input vector $\overline{X} \in \mathcal{D}_f$, if $Z \subseteq set(\overline{X})$ then $f[\overline{X}] = val(Z)$. Clearly, if $V \subseteq W \subseteq set(\overline{X})$ and V is a determining set for f, then W is also a determining set for f.

If a function f satisfies the assumption of Theorem 3.1, then for every vector $\overline{X} \in \mathcal{D}_f$, $set(\overline{X})$ is a determining set. If V and W are determining sets that are included in $set(\overline{X})$, then $val(V) = val(W) = f[\overline{X}]$. This implies the next lemma:

Lemma 3.3 *If a function f satisfies the assumption of Theorem 3.1, then for every input vector $\overline{X} \in \mathcal{D}_f$, each determining set $Z \subseteq set(\overline{X})$ determines the same output $val(Z)$.*

Algorithm 1 Computing a function with bounded registers: code for process p with input x.

```
write(r_x, true);
inputs ← {x};
for i = 1 to ∞ do
    for j = 1 to i do
        v ← read(r_j);
        if v = true then
            inputs ← inputs ∪ {j};
            if IsDetermining(inputs) then return val(inputs);
```

Intuitively, the idea is to find a determining set that allows to compute the function: Each process announces its input and waits to see a determining set. Processes may see different determining sets, but Lemma 3.3 implies that they determine the same output.

We start with a simple protocol using an unbounded number of Boolean registers. The protocol assumes that the set of possible inputs is enumerable and treats them as integers, for simplicity. The protocol uses a Boolean register r_x for every possible input x; all registers are initially *false*. A process announces its input by writing *true* to the appropriate register, and then repeatedly reads the registers (each time more and more registers), until it obtains a determining set Z; then it outputs $val(Z)$.

The code appears in Algorithm 1. The variables *inputs*, i, j and v are local to each process. The predicate IsDetermining returns $true$ if and only if its input is a determining set for f.

Lemma 3.4 *Algorithm 1 computes a function f, if f satisfies the characterization.*

Proof: Let $\overline{X}$ be an input vector. By Lemma 3.3, there is a determining set $Z \subseteq set(\overline{X})$, and every set that includes Z is a determining set, in particular, $set(\overline{X})$ is. Therefore, after all processes have taken one step, the shared memory contains a determining set. Thus, every process eventually sees a determining set and terminates. By Lemma 3.3, every determining set determines the same output and all processes decide on $f[\overline{X}]$. ■

3.2 Computing with two registers

We now present a protocol using only two shared registers. The protocol uses two unbounded registers, *list* and *result*, both initially *empty*. The *list* register is used to collect all the inputs known so far, while the *result* register is used to announce a decision. As long as no decision is announced, a process makes sure that *list* contains all inputs it knows of.

The code appears in Algorithm 2. The variables y, *known* and *inputs* are local to each process.

Algorithm 2 Computing a function with two registers: code for process p with input x.

```
1:  known ← {x};
2:  repeat
3:      y ← read(result);
4:      if y ≠ empty then return val(y);
5:      inputs ← read(list);
6:      known ← inputs ∪ known;
7:      if IsDetermining(known) then                // is known a determining set for f?
8:          write(result, known);
9:          return val(known);
10:     if not (known ⊆ inputs) then write(list, known);
```

Let $known_i(C)$ denote the value of *known* held by process p_i in configuration C.

For a configuration C and a finite schedule σ, if $known_i(\sigma(C))$ contains some input that does not appear in $known_i(C)$ then p_i *learns a new input* in (C, σ).

Lemma 3.5 *Let C be a reachable configuration of Algorithm 2 and let σ be a fair schedule. If for every i, $known_i(C)$ is not a determining set, then there is finite prefix τ of σ such that some process p_j learns a new input in (C, τ).*

Proof: Fix some fair schedule σ. If $known_i(C)$ contains all inputs, for some i, then $known_i(C)$ is a determining set. So assume that for every i, $known_i(C)$ does not contain all inputs.

Let σ' be a prefix of σ such that every process completes at least one full iteration of the repeat loop (lines 2-10) in (C, σ').

Claim 3.6 *There is a write to list in (C, σ').*

Proof: Suppose towards a contradiction that there is no write to *list* in (C, σ') and consider some process p_i. Since p_i completes one full iteration in (C, σ'), p_i reads *list* in Line 5 but does not write to it in Line 10, meaning that the condition in Line 10 is false. Let D_i be the configuration in which p_i checks the condition in Line 10. By the protocol, $known_i(D_i)$, and hence $known_i(C)$, is included in the set that p_i reads from *list*. This holds for every process p_i and since $known_i(C)$ includes p_i's input, it follows that *list* includes the inputs of all processes and is a determining set. Therefore, $known_j(C)$ is a determining set for p_j, the last process writing to *list* before C, which is a contradiction. ■

Since a process does not learn a new input in (C, σ), whenever a process writes, it writes the value of its *known* variable in configuration C to *list*; thus, after *list* is written in σ', it contains the value of $known_k(C)$, for some k. Thus, the value in *list* at every intermediate configuration of $(\sigma'(C), \sigma - \sigma')$ is the value of $known_k(C)$, for some k (possibly different k's for different configurations).

Fix a process p_j for which the set $known_j(C)$ is minimal, in the sense that it does not properly contain any other set $known_i(C)$. Note that if for some process p_i, $known_i(C) \neq known_j(C)$, then $known_i(C)$ contains an input which is not in $known_j(C)$. Thus, we have that if p_j, or another process $p_{j'}$ for which $known_{j'}(C) = known_j(C)$, reads a value $known_i(C)$ where $known_i(C) \neq known_j(C)$, then it learns a new input. Thus we may assume that every such process $p_{j'}$ always reads $known_j(C)$ in $(\sigma'(C), \sigma - \sigma')$, and hence never writes.

Therefore, some process p_i for which $known_i(C) \neq known_j(C)$ eventually writes; henceforth, $known_j(C)$ is never written to *list*. When p_j reads after this write, it reads a value $known_k(C) \neq known_j(C)$, which is a contradiction. ∎

Lemma 3.7 *Algorithm 2 computes a function f if f satisfies the characterization.*

Proof: Let α be a fair execution of the protocol with input vector $\overline{X}$. By repeated application of Lemma 3.5, eventually some process either knows a determining set or learns $set(\overline{X})$, which by Lemma 3.3, includes a determining set. Thus, in some configuration in α some process knows a determining set, writes this set to *result* and terminates. After that, *result* contains a non-empty value, and hence, all processes terminate. By the protocol, only determining sets are written to *result*, and by Lemma 3.3, all processes decide on $f[\overline{X}]$. ∎

3.3 Computing with a single register

We have seen that every function computable in the model, can be computed with two unbounded registers. This section explores which functions can be computed with a single unbounded register. The next lemma shows that some computable functions cannot be computed with a single register.

Lemma 3.8 *Let f be a non-constant function, such that $\overline{X} \in \mathcal{D}_f$ if and only if $\overline{X}$ is an N-entry vector and $\overline{X}$ contains exactly k different entries (for some known constant $k \geq 2$). If $N > k$, then f is not computable with one register.*

Proof: Let f be a non-constant function that satisfies the definition and is computable in the totally anonymous model. Suppose in contradiction that f can be computed with one register.

Since f is computable in the model, Theorem 3.1 implies that the input set uniquely determines the function value. Since f is non-constant, there are two input vectors $\overline{X}$ and $\overline{X'}$ such that $f[\overline{X}] \neq f[\overline{X'}]$. Since f is computable in the model, there exists $x \in set(\overline{X})$ such that $x \notin set(\overline{X'})$. Let x' be a value in $set(\overline{X'})$.

Consider the set $S = set(\overline{X'}) \cup \{x\} - \{x'\}$ and an N-vector $\overline{Y}$ such that $set(\overline{Y}) = S$. $\overline{Y}$ is a legal input vector for f since there are exactly k different values in $\overline{Y}$. We assume that $f[\overline{Y}] \neq f[\overline{X}]$. (If $f[\overline{Y}] = f[\overline{X}]$, then $f[\overline{Y}] \neq f[\overline{X'}]$ and a similar argument holds for $\overline{Y}$ and $\overline{X'}$.)

Consider an execution α of k processes such that the input set in α is S. By Lemma 2.3, the processes decide in α on $f[\overline{Y}]$. It is clear that there exists a write in α (otherwise a process with input x cannot decide correctly). Let p be a process that writes first in α and let a be the input of p in α.

By similar arguments, there is a write in an execution β of k processes in which the input set is $set(\overline{X})$. Let q be a process that writes first in β and let b be the input of q in β.

Suppose that both p and q decide in their solo executions; they must decide on different values—p decides on $f[\overline{Y}]$ and q decides on $f[\overline{X}]$. Consider the following execution: p runs until it is about to take its first write step; then q runs by itself; since p does not write yet, q eventually decides on $f[\overline{X}]$; then p continues to run; in its first step p writes to the only register and thus overwrites any trace of the steps of q; therefore p eventually decides on $f[\overline{Y}]$. In this execution p and q decide on different values, which is a contradiction.

Therefore, one of them does not decide in its solo execution; suppose, without loss of generality, that p does not decide in its solo execution.

Consider an execution γ of $k+1$ processes: k processes that have input set $set(\overline{Y})$ and process p with input a. First, p runs till it ready to write in the first time; then all other processes run; by Lemma 2.3, they eventually decide on $f[\overline{Y}]$. Then, p continues to run and in its first step p writes to the only register and thus overwrites any trace of the other processes. Since p does not decide in its solo execution, p never decides. Since $k+1 \leq N$ and $a \in set(\overline{Y})$, by Lemma 2.3, all processes eventually decide in γ, which is a contradiction. ■

Thus, there is a function computable in the totally anonymous model that cannot be computed with one register. In contrast, there are functions that can be computed with a single register. Algorithm 3 computes a function f, if the number of inputs, N, is known and all inputs are different. In this case, the inputs can be used as process' identifiers.

Clearly, f can be computed by collecting all the inputs. In the protocol, a single register *list* collects the inputs. In the first stage, each process declares its input by adding it to *list*, in a manner similar to Algorithm 2. When all inputs are written to *list*, a leader is chosen; the leader knows the function value. In the second stage, the leader passes the function value to every other process by hand-shaking with it. After hand-shaking with the leader, the process terminates; the leader terminates after hand-shaking with all processes.

By arguments similar to the proof of Lemma 3.5, eventually the process with the minimal input finishes the first phase and starts the hand-shaking. It is easy to see that any non-leader process hand shakes with the leader and terminates. Therefore, the leader also terminates. This implies the correctness of the protocol.

Lemma 3.9 *If f is a function computable in the totally anonymous model such that N, the number of inputs, is known and all inputs are different, then Algorithm 3 computes f.*

Algorithm 3 Computing a function with one register: code for process p with input x.

```
known ← {x};
repeat forever
    inputs ← read(list);
    if for some j: inputs = ack_j or inputs contains token_j then break;
                          // ack_j and token_j are constants, which cannot be inputs
    known ← inputs ∪ known;
    if |known| = N and x is the minimal value in known then break;
    if not (known ⊆ inputs) then write(list, known);

if |known| = N and x is the minimal value in known then          // p is the leader
    for j = 2 to N do
        repeat
            write (list, ⟨known, token_j⟩);
            repeat
                l ← read(list);
            until l ≠ ⟨known, token_j⟩;
        until l = ack_j;
    write(list, ⟨known, token_{N+1}⟩);
else                                                             // p is not a leader
    repeat
        l ← read(list);
    until l contains token_j for some j
    known ← the first field in l
    i ← order of x in known;
    repeat
        repeat
            l ← read(list);
        until l contains token_i or (l contains token_j or l = ack_j for some j > i);
        if l contains token_i then write (list, ack_i);
    until l contains token_j or l = ack_j for some j > i;
```

3.4 Weak computing

A protocol $\mathcal{P}$ *weakly computes a function* f if for every legal input vector $\overline{X} = (x_1, x_2, \ldots, x_N)$ and for every fair execution α, such that a process p_i has an input x_i in α, there is a process p_i and a finite prefix of α in which p_i writes $f[\overline{X}]$ into its local output register. That is, one process knows the function value and terminates but it is possible that some (or even all) other processes do not know the output or do not terminate.

It is clear that a function that can be (strongly) computed in the model, can be weakly computed in it. The other direction is also true—no other function can be weakly computed in the model.

Consider a function f that can be weakly computed in the model. Let $\mathcal{P}$ be a protocol that weakly computes f. Consider protocol $\mathcal{P}'$ that uses, in addition to the shared registers used by $\mathcal{P}$, a register *result*. The code of a process p_i in $\mathcal{P}'$ is as its code in $\mathcal{P}$, but between every pair of steps in $\mathcal{P}$, p_i checks if the *result* register is not empty, and if this is the case, it finishes the algorithm and returns the value it read from *result*. When a process knows the function value according to $\mathcal{P}$, it writes the value to the *result* register. Since $\mathcal{P}$ is a protocol that weakly computes f, in any fair execution there is a process p_i that knows the function value. In $\mathcal{P}'$, this process writes the function value to *result* and, therefore, all processes know the function value and terminate. Therefore, $\mathcal{P}'$ (strongly) computes f.

Lemma 3.10 *A function f can be weakly computed in the model, if and only if it can be computed in it.*

One unbounded shared register suffices for weak computing. Consider a protocol like in Algorithm 2 but without the *result* register. From the proof of Lemma 3.7 it follows that this protocol weakly computes a function f that satisfies the characterization.

Corollary 3.11 *Every function that can be computed in the model, can be weakly computed with one unbounded shared register.*

4 The consensus task

Jayanti and Toueg [11] proved that no protocol solves binary consensus in the totally anonymous model if the memory is not initialized. We now show that consensus can be solved when memory is initialized, starting with binary consensus and then turning to the general case.

4.1 Binary consensus

The protocol for binary consensus uses a procedure ReduceOne that reduces the disagreement in processes' inputs; it returns 0, 1 or $\perp$. ReduceOne guarantees that at least one process returns 0 or 1, that if some process returns $v \in \{0,1\}$ then v was the input of some process, and that either only 1 and $\perp$ are returned or the number of 1'es returned is smaller than the number of 1'es in the input.

If N, an upper bound on the number of processes, is known, then consensus can be solved with $N-1$ copies of ReduceOne. A process starts the first copy of ReduceOne with its input; if it gets $\perp$, then it waits to read a non-empty value from a separate decision register; otherwise, it starts the next copy of ReduceOne with the output from the current copy, and follows the same rules. When a process gets a non-$\perp$ output in the $(N-1)$-st copy of ReduceOne, it writes this value to the decision register and decides on it; a process waiting for a non-empty value in the decision register, decides upon reading this value. It is guaranteed that at least one process participates in every copy of ReduceOne, in particular, the $(N-1)$-st copy. Moreover, the inputs to all copies of ReduceOne are valid, and all processes have the same output in the $N-1$-st copy of ReduceOne.

If the number of processes is unknown, we take an infinite number of copies of ReduceOne. Process p performs the copies of ReduceOne sequentially as in the simple solution, but before starting iteration j with v, p checks if a process with $\neg v$ already started iteration j. If so, p waits to read a non-empty value from the decision register. When completing iteration j with $v \neq \perp$, p checks if a process with $\neg v$ was in iteration $j-1$. If there is no such process, p writes its output from iteration j to the decision register and terminates.

ReduceOne uses two Boolean shared registers, $awake[0]$ and $awake[1]$, both initially *false*. A process p with input x writes *true* to $awake[x]$ and reads the other register. If the other register contains *false*, the procedure returns x to p. If the other register contains *true* and $x=0$, the procedure returns $\perp$ to p. If the other register contains *true* and $x=1$, then the procedure returns 0 to p. The variable w is local to each process.

The code appears in Algorithm 4. The variables a, j, *exist* and d are local to each process. The subscript j denotes the j'th copy of ReduceOne and its registers. Note that the code inspects the internal variables of ReduceOne (Lines 3 and 8).

We start with the properties of ReduceOne.

Lemma 4.1 *Procedure ReduceOne guarantees the following properties:*

Non-triviality: *The output of at least one process is not $\perp$.*

Validity: *If the output of p_i is $v \neq \perp$, then the input of some process is v.*

Monotonicy: *If some process outputs 0 and another process outputs 1, then the number of processes with output 1 is strictly smaller than the number of processes with input 1.*

Proof: Line 18 implies that when all processes have the same input v, they will get v as output, and this satisfies all three properties; so, assume both 0 and 1 are inputs.

Algorithm 4 Binary consensus: code for process p with input x.

1: $a \leftarrow x$;
2: for $j = 1$ to ∞ do
3:     $exist \leftarrow$ read$(awake_j[\neg a])$;
4:     if $exist = true$ then goto Line 12
5:     $a \leftarrow \mathsf{ReduceOne}_j(a)$;
6:     if $a = \perp$ then goto Line 12
7:     if $j > 1$ then
8:         $exist \leftarrow$ read$(awake_{j-1}[\neg a])$;
9:         if $exist = false$ then
10:             write$(decision, a)$;
11:             decide a;

12: repeat
13:     $d \leftarrow$ read$(decision)$;
14: until $d \neq empty$;
15: decide d;

procedure $\mathsf{ReduceOne}(x\colon 0..1)$: returns $0, 1, \perp$
16: write$(awake[x], true)$;
17: $w \leftarrow$ read$(awake[\neg x])$;
18: if $w = false$ then return x;
19: else   // $w = true$
20:     if $x = 0$ then return $\perp$;
21:     else return 0   // $x = 1$

Non-triviality: A process with input 1 gets output 1 in Line 18 and output 0 in Line 21. Thus, it never gets output $\perp$.

Validity: A process with input 0 gets output 0 in Line 18 and output $\perp$ in Line 20. A process with input 1 gets output 0 in Line 21 only if it reads *true* from *awake*[0], implying that some process with input 0 takes a step.

Monotonicy: Processes with input 0 never get output 1 (Lines 18 and 20). Therefore, we should show that if there are both 0 and 1 in the outputs, then a process with input 1 decides on 0. If all processes with input 1 read *false* from *awake*[0], then they output 1 (by Line 18). In this case, all processes with input 0 read *true* from *awake*[1] and output $\perp$ (Line 20). Otherwise, some process with input 1 reads *true* from *awake*[0] and outputs 0 (Line 21). ■

We now turn to show that Algorithm 4 solves binary consensus.

Lemma 4.2 *For every $k \geq 1$, if a process p_i has input v in the k'th copy of* $\mathsf{ReduceOne}$*, then v is an input of some process.*

Proof: The proof is by induction on k.

Basis: $k = 1$. The input of p_i in $\mathsf{ReduceOne}_1$ is its input to the consensus protocol (Line 1).

Inductive step: Suppose the lemma holds for k. Process p_i participates in $\mathsf{ReduceOne}_{k+1}$ only if it has a non-$\perp$ output from $\mathsf{ReduceOne}_k$. The validity property of ReduceOne implies that p_i's output from $\mathsf{ReduceOne}_k$ is the input of some process to $\mathsf{ReduceOne}_k$. By the inductive hypothesis, the input of a process in $\mathsf{ReduceOne}_k$ is the input of some process to the consensus protocol, which proves the lemma. ∎

Lemma 4.3 *For every k, $1 \leq k \leq N$, either all processes participating in $\mathsf{ReduceOne}_k$ have the same input or there are at most $N - k$ processes with input 1.*

Proof: The proof is by induction on k.

Basis: $k = 1$. If all processes have input 1 in $\mathsf{ReduceOne}_1$, then the lemma follows. Otherwise, some process has input 0; hence, at most $N - 1$ processes have input 1, and the lemma follows.

Inductive step: Suppose the lemma holds for k. The input of p_i in $\mathsf{ReduceOne}_{k+1}$, v, is its output from $\mathsf{ReduceOne}_k$ (Line 5). If all processes have input v in $\mathsf{ReduceOne}_k$, then by the validity of ReduceOne all processes that get an non-$\perp$ output, return v. Therefore, all processes have the same input in iteration $k + 1$. If there are processes with input 0 and processes with input 1 in $\mathsf{ReduceOne}_k$, then by the inductive hypothesis, at most $N - k$ processes have input 1. By the monotonicity of ReduceOne, either all processes that get a non-$\perp$ output get the same output in $\mathsf{ReduceOne}_k$, in which case, the lemma clearly holds, or the number of processes with output 1 is less than the number of processes with input 1. In the later case, the number of processes with output 1 is at most $N - k - 1$, that is, the number of processes with input 1 in $\mathsf{ReduceOne}_{k+1}$ is at most $N - (k + 1)$, as needed. ∎

We say that a process *decides in iteration* k if it decides in Line 11 with $j = k$.

Lemma 4.4 *For every $k \geq 1$, if no process decides in iteration k, then at least one process participates in the $k + 1$'st copy of ReduceOne.*

Proof: Let P be the set of processes participating in iteration k, and assume that no process in P decides in iteration k. By the non-triviality property of ReduceOne, at least one process from P gets a non-$\perp$ output in $\mathsf{ReduceOne}_k$, and starts iteration $k+1$, by reading in Line 3. Let p be the first process to read in Line 3 of iteration $k+1$. Clearly, no process writes to registers of iteration $k + 1$ before this read and therefore, p reads *false* in Line 3 and participates in $\mathsf{ReduceOne}_{k+1}$. ∎

Lemma 4.5 *For every $k \geq 2$, if some process decides on v in iteration k, then no process with input $\neg v$ participates in iteration k.*

Proof: Let p be a process which decides on v in Line 11 of iteration k. A process with input x starts $\mathsf{ReduceOne}_{k-1}$ with a write to $awake_{k-1}[x]$ (Line 16); since p reads false from $awake_{k-1}[\neg v]$, no process with input $\neg v$ starts $\mathsf{ReduceOne}_{k-1}$ before p starts $\mathsf{ReduceOne}_k$. Therefore, if a process with input $\neg v$ finishes $\mathsf{ReduceOne}_{k-1}$, it will read true in Line 3 of iteration k and will not participate in $\mathsf{ReduceOne}_k$. ■

Lemma 4.6 *Some process decides in iteration $k \leq N+1$.*

Proof: By Lemma 4.4 for $k = N$, if no process decides in iterations $1, \ldots, N$, then some process participates in iteration $N+1$. Lemma 4.3 implies that all processes have the same input v in $\mathsf{ReduceOne}_N$. By the validity of $\mathsf{ReduceOne}$, all processes output v in $\mathsf{ReduceOne}_N$ and in $\mathsf{ReduceOne}_{N+1}$. Since there is no process with input $\neg v$ in $\mathsf{ReduceOne}_N$, the read in Line 8 of iteration $N+1$ returns *false* to every participating process. Therefore, every process participating in iteration $N+1$ decides. ■

Theorem 4.7 *Algorithm 4 solves binary consensus.*

Proof: Lemma 4.2 implies the validity of the protocol. By Lemma 4.6, some process decides at iteration $N+1$ at most. Let k be the first iteration in which some process decides; let v be its decision. By Lemma 4.5, all processes that decide in an iteration $\geq k$ decide on v. These processes write v to *decision* in Line 10. Therefore, every process executing Lines 12-14 (waiting for a decision), eventually reads v from *decision* and decides on v. ■

Algorithm 4 has at most $N+1$ iterations, each using two shared registers; therefore, the protocol uses $O(N)$ shared registers. A process takes at most five steps in every iteration. Since there are at most $N+1$ iterations, a process takes $O(N)$ steps in the protocol.

4.2 General consensus

Non-binary consensus is achieved by agreeing (using binary consensus) on each bit of the output. We first consider the case where the inputs are from a known domain, $0, 1, \ldots, K-1$.

Denote $l = \lceil \log K \rceil$, the length of the binary representation of $K-1$. The protocol for K-valued consensus uses l binary consensus protocols. In addition, there is a $K+1$-valued register *decision*, which can contain values from $\{0, \ldots, K-1, empty\}$; it is initially *empty*. Each binary consensus protocol determines one bit of the decision value.

A process p with input $x = b_{l-1} \ldots b_1 b_0$ starts iteration 0 with input b_0. If p participates in iteration j and the decision in this iteration is not b_j then p does not participate in later iterations and waits for a non-empty value in *decision*. If p participates in the last iteration, it writes the bits decided in all iterations to *decision*.

The code for K-valued consensus appears in Algorithm 5. The variables d and j are local to each process. BinaryConsensus_j denotes the j'th copy of Algorithm 4, for binary consensus.

Algorithm 5 K-valued consensus: code for process p with input $x = b_{l-1} \ldots b_1 b_0$.

1: for $j = 0$ to $l - 1$ do
2:     $d_j \leftarrow \text{BinaryConsensus}_j(b_j)$;
3:     if $d_j \neq b_j$ then goto Line 6
4: write($decision, d$);
5: decide d;

6: repeat
7:     $d \leftarrow \text{read}(decision)$;
8: until $d \neq empty$;
9: decide d;

Lemma 4.8 *For every k, $0 \leq k \leq l - 1$, some process participates in iteration k.*

Proof: The proof is by induction on k.

Basis: $k = 0$. Clearly, all processes participate in iteration 0.

Inductive step: Suppose the lemma holds for k, and some process participates in iteration k. Clearly, the decision in BinaryConsensus_k is the input of some process participating in BinaryConsensus_k; this process (at least) participates in iteration $k + 1$ (Line 3). ■

By the code (Line 3), a process p with input $x = b_0, b_1, \ldots, b_{l-1}$ participates in iteration $k + 1$ only if the decision in iteration k is b_k. The following lemma follows by induction on k.

Lemma 4.9 *For every k, $0 \leq k \leq l - 1$, a process with input $x = b_0, b_1, \ldots, b_{l-1}$ participates in iteration k only if the decisions in iterations $0, 1, \ldots, k - 1$ are $b_0, b_1, \ldots, b_{k-1}$, respectively.*

By Lemma 4.8, there are processes which participate in the last ($l - 1$'st) iteration. By Lemma 4.9, the decisions in iterations $0, 1, \ldots, l-1$ compose the input of one of these processes, p_i. Thus, all processes terminate and decide on the same value, which is the input of p_i. Therefore, we have the following theorem:

Theorem 4.10 *Algorithm 5 solves K-valued consensus.*

To handle unbounded inputs, the above protocol is modified so that every iteration consists of two stages: In the first stage, as before, the processes solve binary consensus for the j'th bit of the input. In the second stage, processes decide whether to terminate the protocol, using binary consensus. A process p with an input $x = b_{l-1} \ldots b_1 b_0$ participates in the second stage of iteration j, if the outputs in the first stages of previous iterations are $b_j, \ldots, b_1, b_0$; p's input to the second stage is 1, if all bits in x after j are zero (there are no more significant bits in x) and 0, otherwise. If the decision in the second stage is 1, then the processes decide on $d_j \ldots d_1 d_0$. The code appears in Algorithm 6; its correctness proof is similar to the proof of Algorithm 5, and we do not repeat it here.

Algorithm 6 General consensus: code for process p with input $x = b_{l-1} \ldots b_1 b_0$.

1: for $j = 0$ to ∞ do
2: $\quad d_j \leftarrow \text{BinaryConsensus}_j^1(b_j)$;
3: $\quad$ if $d_j \neq b_j$ then goto Line 10
4: $\quad$ if $b_{j+1}, b_{j+2}, \ldots$ are all zeros then $end \leftarrow 1$;
5: $\quad$ else $end \leftarrow 0$;
6: $\quad last \leftarrow \text{BinaryConsensus}_j^2(end)$;
7: $\quad$ if $last = 1$ then
8: $\qquad \text{write}(decision, d_j \ldots d_0)$;
9: $\qquad$ decide $d_j \ldots d_0$;

10: repeat
11: $\quad d \leftarrow \text{read}(decision)$;
12: until $d \neq empty$;
13: decide d;

5 Computing relations

In this section, we characterize the relations that can be computed in our model.

Theorem 5.1 *A relation R can be computed in the totally anonymous model if and only if for every input vector $\overline{X}$,*

$$\bigcap \{R[\overline{Y}] \mid \overline{Y} \in \mathcal{D}_R \text{ and } set(\overline{Y}) \supseteq set(\overline{X})\} \neq \phi.$$

The consensus task satisfies this condition and, as we have seen, it can be computed in our model. The characterization of computable functions (Theorem 3.1) is a special case of Theorem 5.1, when R is a function.

We start with the necessary condition.

Lemma 5.2 *Given a relation R, if there is an input vector $\overline{X}$ such that*

$$\bigcap \{R[\overline{Y}] \mid \overline{Y} \in \mathcal{D}_R \text{ and } set(\overline{Y}) \supseteq set(\overline{X})\} = \phi,$$

then R cannot be computed in the model.

Proof: Suppose for contradiction that $\mathcal{P}$ is a protocol for computing R. Let $\overline{X}$ be an input vector satisfying the conditions of the lemma. Let $set(\overline{X})$ be $\{x_1, x_2, \ldots, x_k\}$.

By Lemma 2.3, in every fair execution with input vector $(x_1, x_2, \ldots, x_k)$, all processes decide on $z \in R[\overline{X}]$. Let C_0^k be an initial configuration with processes $q_1, \ldots, q_k$, where q_i has input

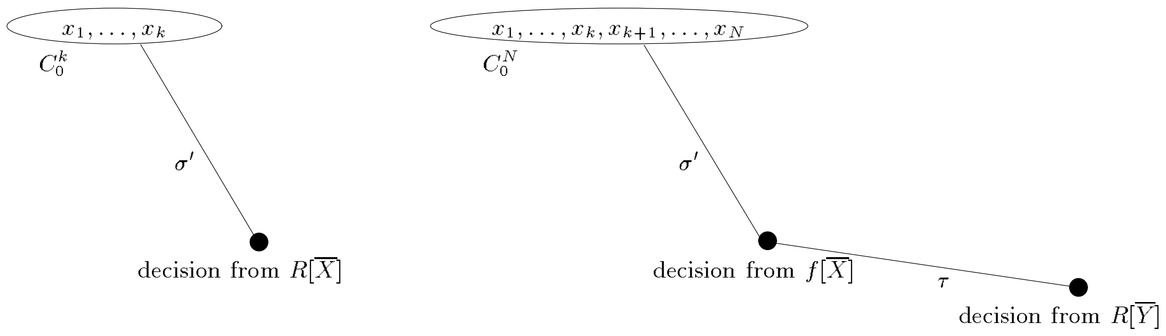

Figure 2: Illustration for the proof of Lemma 5.2.

x_i. Let σ be a fair schedule of $q_1, \ldots, q_k$ and let σ' be a finite prefix of σ such that $q_1, \ldots, q_k$ decide on $z \in R[\overline{X}]$ in (C_0^k, σ').

There exists $\overline{Y} \in \mathcal{D}_R$ such that $set(\overline{Y}) \supseteq set(\overline{X})$ and $z \notin R[\overline{Y}]$. Otherwise, z is in $\bigcap\{R[\overline{Y}] \mid \overline{Y} \in \mathcal{D}_R \textit{ and } set(\overline{Y}) \supseteq set(\overline{X})\}$, contradicting the assumption of the lemma.

Without loss of generality, we can write $\overline{Y}$ as $(x_1, x_2, \ldots, x_k, x_{k+1}, \ldots, x_N)$. Let C_0^N be an initial configuration with processes $q_1, \ldots, q_N$, where q_i has input x_i. $(C_0^k, \sigma') \preceq (C_0^N, \sigma')$, and therefore, all processes $q_1, \ldots, q_k$ decide on z in (C_0^N, σ'). See Figure 2.

Let τ be a fair schedule of processes $q_1, \ldots, q_N$. Consider an execution $\gamma = (C_0^N, \sigma' \cdot \tau)$. The input vector in γ is $\overline{Y}$ and γ is a fair execution, therefore, every process q_i should decide on a value from $R[\overline{Y}]$. But $q_1, \ldots, q_k$ decide on $z \notin R[\overline{Y}]$ in γ, which is a contradiction. ■

Lemma 5.2 generalizes the corresponding lemma for functions (Lemma 3.2). In Lemma 5.2, we take a vector $\overline{X}$ and an execution with decision z, and only then choose a vector $\overline{Y}$ such that $z \notin R[\overline{Y}]$. Here it is not possible to choose the two vectors arbitrarily as in Lemma 3.2, because the outputs for an input vector can be different in different executions.

To prove that the condition is sufficient, consider a relation R satisfying the assumption of Theorem 5.1.

The *tower* of $set(\overline{X})$ is $\{set(\overline{Y}) \mid \overline{Y} \in \mathcal{D}_R$ and $set(\overline{Y}) \supseteq set(\overline{X})\}$. For every input vector $\overline{X}$, the set $set(\overline{X})$ is in some tower; Figure 3 presents the maximal towers for the domain $\{0, 1, 2\}^*$. There are three towers: The tower of all input sets that contain 0, the tower of inputs that contain 1 and the tower of inputs that contain 2. Note that the tower of a non-minimal input set is included in some tower of a minimal input set, and thus it can be omitted. For example, the tower induced by input set $\{0, 1\}$ is $\{\{0, 1\}, \{0, 1, 2\}\}$, and it is included in the tower of 0 and the tower of 1.

0	1	2
0 1	0 1	0 2
0 2	1 2	1 2
0 1 2	0 1 2	0 1 2
tower of 0	tower of 1	tower of 2

Figure 3: Towers for the domain $\{0, 1, 2\}^*$.

By the assumption on R, each tower has an output which is legal for all inputs in the tower. Thus, if all processes agree on a tower to which the input vector belongs, they can compute the relation. If a subset of the input set belongs to some tower, then the set belongs to the same tower; thus, it suffices to agree on some set in the tower. The idea of the solution is to reach consensus on some subset of the input set, which includes the input set of a legal input vector, and hence it determines the output.

A set Z is a *valid subset*, if there is an input vector $\overline{X} \in \mathcal{D}_R$ such that $Z \supseteq set(\overline{X})$. A valid subset, Z, can be mapped in some pre-determined way to a tower which contains Z, from which an output can be computed in a unique way. Thus, once the valid subset is determined, the output can be locally computed by every process.

The protocol determines a valid subset for a relation R iteratively; in every iteration, a different input is picked, until the set of chosen values is a valid subset. A consensus protocol (shown in Section 4) is used in every iteration to agree on the input picked. Note that in different executions, different valid subsets (and hence outputs) may be found.

In addition to the registers used by the consensus protocols, the protocol uses an unbounded register, *agreement*, initialized *empty*. Every process p_i acts according to the following rules.

- p_i starts the first iteration with its input;
- if the decision in the j'th consensus protocol is p_i's input, then p_i does not participate in later iterations;
- if the decisions in the first j iterations form a valid subset, and p_i participates in iteration j, then p_i writes the subset to the *agreement* register.
- if p_i stops participating at some stage, then it reads the *agreement* register until it obtains a non-empty value.
- p_i decides on an output determined by a value in *agreement*.

Lemma 5.3 *The above protocol computes a relation R, if R satisfies the characterization.*

Proof: An input is the decision of a single iteration only, since all processes with this input do not participate in later iterations. Therefore, the number of iterations in the protocol is bounded by number of different inputs, which is at most N. The set of all processes' inputs is a valid subset, and in every iteration some new input is chosen, hence, each process eventually decides on some valid subset and terminates. All processes compute the same valid subset, since they agree on the same input in each iteration; thus, they decide on the same output. ■

6 Lower bounds for binary consensus

6.1 Number of registers

Fix a consensus protocol for N processes. The next definitions follow Fischer, Lynch and Paterson [8]. A configuration C is *0-valent* if only 0 is decided on in an execution from C; similarly, C is *1-valent* if only 1 is decided on in an execution from C; C is *univalent* if it either 0-valent or 1-valent; otherwise, it is *bivalent*. If C is bivalent, then there is an execution from C with decision 0 and an execution from C with decision 1.

Although processes do not fail, Lemma 2.3 can be used to show:

Lemma 6.1 *There is an initial configuration with two processes which is bivalent.*

Proof: Consider an initial configuration, C_0 of two processes, p_0 with input 0 and p_1 with input 1. $\overline{X} = (0, \ldots, 0)$ is an N-entry input vector for consensus; Lemma 2.3 implies that p_0 decides on 0 in an infinite p_0-only execution from C_0. $\overline{Y} = (1, \ldots, 1)$ is an N-entry input vector for consensus; Lemma 2.3 implies that p_1 decides on 1 in an infinite p_1-only execution from C_0. Therefore, C_0 is bivalent. ■

Following Burns and Lynch [5], we say that process p *covers* a register r in a configuration C if the step of p from C is a write to r. We say that r *is covered* by p in C.

In order to prove the lower bound on the number of registers, we iteratively construct executions such that in each iteration, one more register is covered.

A process is either *active* in an iteration, i.e., take steps in the iteration, or *waiting*, i.e., take no steps in the iteration. In every iteration, waiting processes cover all registers written in previous iterations. If no new register is written, the waiting processes can overwrite all registers and continue execution as if the current iteration never happened. Thus, an additional register must be written, and the execution can be extended with one more covered register. The next lemma proves this property and is the inductive step of the lower bound proof.

Lemma 6.2 *Let α be a finite execution of $n \leq N/2$ processes, such that $C_{end}(\alpha)$ is bivalent, k registers are covered in $C_{end}(\alpha)$ and all writes in α are to the covered registers. There is a finite execution β of $2n$ processes, such that $C_{end}(\beta)$ is bivalent, $k+1$ registers are covered in $C_{end}(\beta)$ and all writes in β are to the covered registers.*

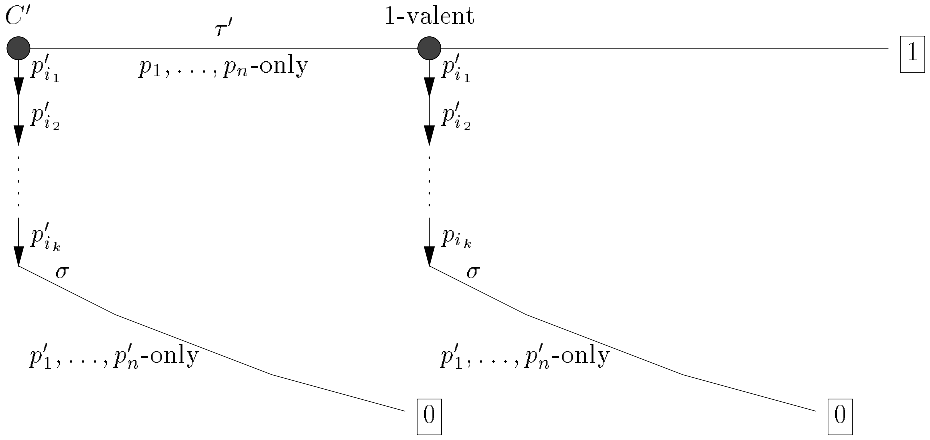

Figure 4: Illustration for the proof of Lemma 6.2.

Proof: Assume α is a finite execution of $q_1, \ldots, q_n$ satisfying the conditions of the lemma, and let α' be the multiplication of α by $\{\{p_1, p'_1\}, \ldots, \{p_n, p'_n\}\}$. Denote $C = C_{end}(\alpha)$ and $C' = C_{end}(\alpha')$.

By Lemma 2.1, $\alpha|q_i = \alpha'|p'_i$, for every i, and all writes in α' are to the k covered registers, denoted $r_1, \ldots, r_k$. Thus, there are k processes, $p'_{i_1}, \ldots, p'_{i_k}$ in $C_{end}(\alpha')$ that cover $r_1, \ldots, r_k$.

Consider a $\{p'_1, \ldots, p'_n\}$-only schedule σ starting with the sequence $p'_{i_1} \cdot p'_{i_2} \cdots p'_{i_k}$; that is, every covering process takes one step in the first k steps of σ. The n inputs of processes $p'_1, \ldots, p'_n$ can be easily extended to an N-vector of inputs for consensus. By Lemma 2.3, $p'_1, \ldots, p'_n$ decide in (C', σ). Assume that the decision in (C', σ) is 0.

Since C is bivalent, there is a $\{q_1, \ldots, q_n\}$-only schedule ρ such that 1 is decided in (C, ρ). Since $\alpha|q_i = \alpha'|p_i$ for every i, it follows that there is a $\{p_1, \ldots, p_n\}$-only schedule τ such that 1 is decided in (C', τ).

Assume there is a prefix τ' of τ such that $\tau'(C')$ is univalent and the only writes in (C', τ') are to $r_1, \ldots, r_k$. Since the decision in (C', τ) is 1, $\tau'(C')$ must be 1-valent. Consider the execution $(\tau'(C'), \sigma)$. Since all writes in (C', τ') are to $r_1, \ldots, r_k$, the execution $(\tau'(C'), \sigma)$, after the k writes to $r_1, \ldots, r_k$, is the same as (C', σ). Therefore, the decision in $(\tau'(C'), \sigma)$ is also 0, contradicting fact that $\tau'(C')$ is 1-valent (see Figure 4.)

Therefore, there must be a write to a register other than $r_1, \ldots, r_k$ in (C', τ) and the configuration before the first such write is bivalent. Let τ'' be the longest prefix of τ that does not include the first write to $r_{k+1} \notin \{r_1, \ldots, r_k\}$. Let β be the execution extending α' with the schedule τ''. In β, $2n$ processes participate; $C_{end}(\beta)$ is bivalent; $k+1$ registers are covered in $C_{end}(\beta)$: $r_1, \ldots, r_k$ are covered by $p'_{i_1}, \ldots, p'_{i_k}$ and by construction, r_{k+1} is also covered; all writes in β are to $r_1, \ldots, r_{k+1}$. Thus, β is the desired execution. ■

Lemma 6.3 *There is an execution α_k of 2^{k+1} processes in which k registers are covered, for every k, $0 \leq k < \log N$.*

Proof: We prove that for every k, $0 \leq k < \log N$, there is an execution α_k of 2^{k+1} processes such that $C_{end}(\alpha_k)$ is bivalent, k registers are covered in $C_{end}(\alpha_k)$ and all writes in α_k are to the k covered registers. The proof is by induction on k.

Basis: $k = 0$. By Lemma 6.1, there is a bivalent initial configuration C_0 with two processes. The empty execution of two processes from C_0, denoted α_0, has the required properties.

Inductive step: Suppose the lemma holds for $k-1$, where $1 \leq k < \log N$. That is, there is an execution α_{k-1} of 2^k processes such that $C_{end}(\alpha_{k-1})$ is bivalent, $k-1$ registers are covered in $C_{end}(\alpha_{k-1})$ and all writes in α_{k-1} are to the $k-1$ covered registers. By Lemma 6.2, there is an execution α_k of $2 \cdot 2^k = 2^{k+1}$ processes, such that $C_{end}(\alpha_k)$ is bivalent, k registers are covered in $C_{end}(\alpha_k)$ and all writes in α_k are to the k covered registers. ■

Taking $k = \log N - 1$ in Lemma 6.3, we get:

Theorem 6.4 *In the totally anonymous model, a protocol solving consensus among N processes requires $\log N - 1$ shared registers.*

6.2 Round complexity

Given a (finite) schedule σ of processes $p_1, \ldots, p_n$, $round_1(\sigma)$ is the minimal prefix of σ in which every process p_i appears, if such a prefix exists, and σ, otherwise. If some process does not appear in $round_1(\sigma)$, then $round_1(\sigma)$ is *incomplete*, otherwise $round_1(\sigma)$ is *complete*. If for a given schedule σ, $round_1(\sigma), \ldots, round_k(\sigma)$ are defined and $round_k(\sigma)$ is complete, then $round_{k+1}(\sigma)$ is $round_1(\sigma - round_1(\sigma) - \cdots - round_k(\sigma))$. That is, $round_{k+1}(\sigma)$ is $round_1$ of the suffix of σ without the k first rounds.

The *number of rounds* in σ is the maximal k such that $round_k(\sigma)$ is complete.

For an execution $\alpha = (C_0, \sigma)$, $round_1(\alpha), \ldots, round_k(\alpha)$ are induced in a natural way from the rounds of σ. The *number of rounds* in α is the number of rounds in the schedule corresponding to the maximal finite prefix of α in which no process decides.

The *round complexity of a protocol* is the maximal number of rounds in its executions.

For a schedule σ of $p_1, \ldots, p_n$ with k rounds, the set of processes that do not appear in $round_k(\sigma)$ is denoted $NY_k(\sigma)$ (*not yet* in round k); $NY_k(\sigma)$ is empty if $round_k(\sigma)$ is complete.

The next simple lemma shows that multiplication (defined in Section 2.3) preserves the number of rounds and the NY sets.

Lemma 6.5 *Let σ be a schedule of $q_1, q_2, \ldots, q_n$ with k complete rounds and let τ be the multiplication of σ by $\{P_1, \ldots, P_n\}$. Then τ has k complete rounds (with respect to processes $\cup_{i=1}^n P_i$) and for every $p \in P_i$, if $p \in NY_{k+1}(\tau)$ then $q_i \in NY_{k+1}(\sigma)$.*

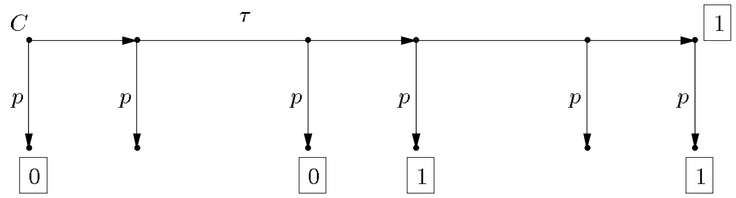

Figure 5: Steps of p lead to different valence.

Proof: For every t, $1 \leq t \leq k$, $round_t(\sigma)$ is complete, and thus every process q_i appears at least once in $round_t(\sigma)$. Therefore, the multiplication of $round_t(\sigma)$ is a complete round, since every process in the group of q_i appears at least once. Thus, the multiplication of $round_1(\sigma) \cdot round_2(\sigma) \cdots round_k(\sigma)$ by $\{P_1, \ldots, P_n\}$ includes k complete rounds. A process $p \in P_i$ appears in the multiplication of $round_{k+1}(\sigma)$ if and only if q_i appears in $round_{k+1}(\sigma)$. Therefore, if $p \in NY_{k+1}(\tau)$ then $q_i \in NY_{k+1}(\sigma)$. ∎

A configuration C is *critical* if there are two processes p_i and p_j such that a step of p_i in C leads to a v-valent configuration, while a step of p_j followed by a step of p_i leads to a $\neg v$-valent configuration. That is, the step p_i and the steps $p_j p_i$ lead to configurations with the opposite valences. We say that p_i is a *critical leader in* C. The next lemma was used in [6, 8, 19].

Lemma 6.6 *Let C be a bivalent configuration and let p be a process. Then there is a finite schedule σ such that either $\sigma \cdot p(C)$ is bivalent or p is a critical leader in $\sigma(C)$.*

Proof: If $p(C)$ is bivalent then take σ to be the empty schedule. Otherwise, $p(C)$ is univalent; without loss of generality, $p(C)$ is 0-valent. Since C is bivalent, there is a finite schedule τ such that $\tau(C)$ is 1-valent. Clearly, $\tau \cdot p(C)$ is 1-valent. We consider what happens if p takes a step after each prefix of τ (see Figure 5).

The lemma clearly holds if a step of p leads to a bivalent configuration after some prefix of τ. So assume that a step of p leads to a univalent configuration from every configuration in (C, τ). Since $p(C)$ is 0-valent while $\tau \cdot p(C)$ is 1-valent, there are two consecutive configurations in (C, τ), denoted C_0 and C_1, such that a step of p from C_0 leads to 0-valence and a step of p from C_1 leads to 1-valence. The prefix of τ leading to C_0 is the desired schedule. ∎

Let $\mathcal{P}$ be a binary consensus protocol for N processes. We iteratively construct executions of $\mathcal{P}$, each iteration with one more round; all executions end with a bivalent configuration.

Lemma 6.7 *Let α be an execution of $n < N$ processes, that includes exactly k complete rounds (for the n processes), such that $C_{end}(\alpha)$ is bivalent. Then there is an execution β of at most $n+1$ processes that includes k complete rounds, such that $C_{end}(\beta)$ is bivalent and $|NY_{k+1}(\beta)| < |NY_{k+1}(\alpha)|$.*

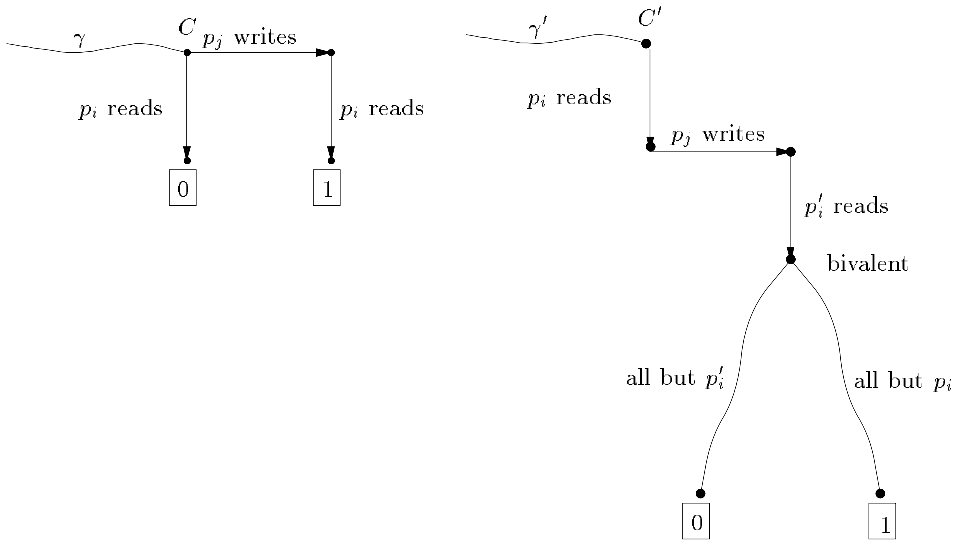

Figure 6: Proof of Lemma 6.7: p_i reads r and p_j writes to r.

Proof: Since $C_{end}(\alpha)$ is bivalent and α includes exactly k complete rounds, the $k+1$'st round of α is incomplete; that is, $NY_{k+1}(\alpha)$ is not empty. Consider some process $p_i \in NY_{k+1}(\alpha)$. By Lemma 6.6, there is a finite schedule τ such that either a step of p_i in $\tau(C_{end}(\alpha))$ leads to a bivalent configuration or p_i is a critical leader in $\tau(C_{end}(\alpha))$. Denote (α, τ) by γ.

If a step of p_i in $C_{end}(\gamma)$ leads to a bivalent configuration, then let $\beta = (\gamma, p_i)$. All requirements hold: n processes participate in β; $C_{end}(\beta)$ is bivalent, by assumption; β is an extension of α and therefore includes k complete rounds; p_i is not in $NY_{k+1}(\beta)$, and therefore it includes at least one process less than $NY_{k+1}(\alpha)$. Therefore, β is the desired execution.

Otherwise, $C = C_{end}(\gamma)$ is a critical configuration and p_i is a critical leader in C, say, together with process p_j. Note that, since $p_i \neq p_j$, p_i takes the same step in both executions.

Consider what are the next steps of p_i and p_j from C. If p_i and p_j access different registers or both p_i and p_j read, then $(C, p_i \cdot p_j) \approx (C, p_j \cdot p_i)$. This contradicting the fact that $p_i \cdot p_j(C)$ is 0-valent and $p_j \cdot p_i(C)$ is 1-valent. Therefore, p_i and p_j access the same register r and at least one of them writes to r. There are two cases according to the operation applied by p_i.

Case 1: p_i reads from r and p_j writes to r (Figure 6). Consider an execution γ' which is the multiplication of γ by $\{\{p_1\}, \ldots, \{p_i, p_i'\}, \ldots, \{p_j\}, \ldots, \{p_n\}\}$. By Corollary 2.2, $\gamma' \approx \gamma$. Denote $C' = C_{end}(\gamma')$ and look at the schedule $\pi = p_i \cdot p_j \cdot p_i'$.

Since $\gamma'|p_i = \gamma|p_i$, $\gamma'|p_i' = \gamma|p_i$ and $\gamma'|p_j = \gamma|p_j$ and the step of p_i' is read, it follows that $(C, p_i \cdot p_j) \preceq (C', p_i \cdot p_j \cdot p_i')$. Since $p_i \cdot p_j(C)$ is 0-valent, if p_i' does not take a step in a schedule σ', then 0 is decided in $(\pi(C'), \sigma')$. Since $\gamma'|p_i = \gamma|p_i$, $\gamma'|p_i' = \gamma|p_i$, $\gamma'|p_j = \gamma|p_j$, and the step

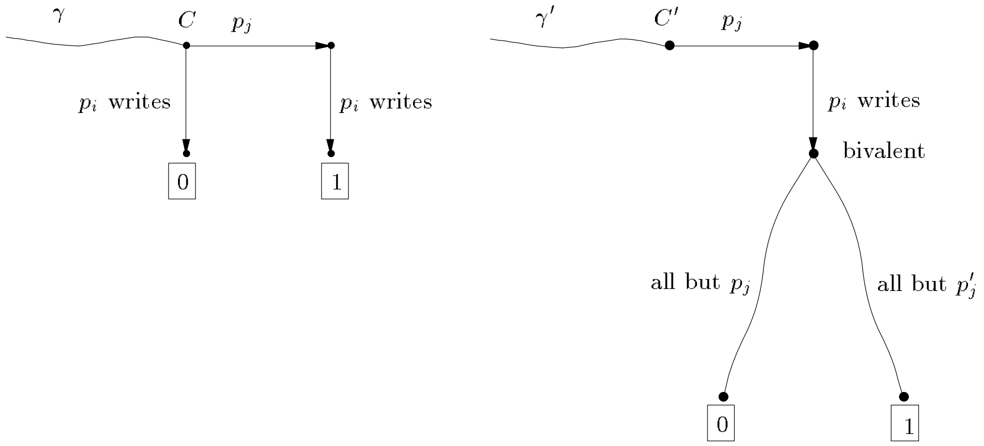

Figure 7: Proof of Lemma 6.7: p_i writes to r.

of p_i is read, it follows that $(C, p_j \cdot p_i) \preceq (C', p_i \cdot p_j \cdot p_i')$. Since $p_j \cdot p_i(C)$ is 1-valent, if p_i does not take a step in a schedule σ'', then 1 is decided in $(\pi(C'), \sigma'')$. Therefore, $\pi(C')$ is bivalent.

Denote $\beta = (\gamma', \pi)$; $n + 1$ processes participates in β. $C_{end}(\beta)$ is bivalent and thus no process decides in β. Lemma 6.5 implies that γ' includes k complete rounds, since γ includes k complete rounds. Therefore, p_i, p_i', p_j are not in $NY_{k+1}(\beta)$. For every other process p, by Lemma 6.5, if $p \in NY_{k+1}(\gamma')$ then $p \in NY_{k+1}(\gamma)$ and thus $p \in NY_{k+1}(\alpha)$. Since $p_i \in NY_{k+1}(\alpha)$, $|NY_{k+1}(\beta)| < |NY_{k+1}(\alpha)|$.

Case 2: p_i writes to r (Figure 7). Consider an execution γ' that is the multiplication of γ by $\{\{p_1\}, \ldots, \{p_i\}, \ldots, \{p_j, p_j'\}, \ldots, \{p_n\}\}$. By Corollary 2.2, $\gamma' \approx \gamma$. Denote $C' = C_{end}(\gamma')$, and look at the schedule $\pi = p_j \cdot p_i$.

Since $\gamma'|p_i = \gamma|p_i$, $\gamma'|p_j = \gamma|p_j$, $\gamma'|p_j' = \gamma|p_j$, the step of p_i is write to r, and p_j accesses r as well, it follows that $(C, p_i) \preceq (C', p_j \cdot p_i)$. Since $p_i(C)$ is 0-valent, if p_j does not take a step in a schedule σ', then 0 is decided in $(\pi(C'), \sigma')$. Since $\gamma'|p_i = \gamma|p_i$ and $\gamma'|p_j = \gamma|p_j$, it follows that $(C_{end}(\gamma), p_j \cdot p_i) \preceq (C_{end}(\gamma'), p_j \cdot p_i)$. Since $p_j \cdot p_i(C)$ is 1-valent, if p_j' does not take a step in a schedule σ'', then 1 is decided in $(\pi(C_{end}(\gamma)), \sigma'')$. Therefore, $\pi(C')$ is bivalent.

Denote $\beta = (\gamma', \pi)$; $n+1$ processes participate in β. $C_{end}(\beta)$ is bivalent and thus no process decides in β. By Lemma 6.5, γ' includes k complete rounds, since γ includes k complete rounds. Therefore, p_i, p_j are not in $NY_{k+1}(\beta)$. By Lemma 6.5, for every process $p \neq p_i, p_j, p_j'$, if $p \in NY_{k+1}(\gamma')$ then $p \in NY_{k+1}(\gamma)$, and hence $p \in NY_{k+1}(\alpha)$. If $p_j \notin NY_{k+1}(\alpha)$ then $p_j' \notin NY_{k+1}(\beta)$; therefore, since $p_i \in NY_{k+1}(\alpha)$, $|NY_{k+1}(\beta)| < |NY_{k+1}(\alpha)|$. If $p_j \in NY_{k+1}(\alpha)$, then it is possible that $p_j' \in NY_{k+1}(\beta)$. However, both p_i and p_j are in $NY_{k+1}(\alpha)$, but not in $NY_{k+1}(\beta)$; hence, $|NY_{k+1}(\beta)| < |NY_{k+1}(\alpha)|$ (one process is added, but two processes are removed). ∎

This lemma is used (within an induction) to prove the next lemma:

Lemma 6.8 *Let α be an execution of $n \leq N/2$ processes with exactly k complete rounds such that $C_{end}(\alpha)$ is bivalent. Then there is an execution β of at most $2n$ processes with $k+1$ complete rounds such that $C_{end}(\beta)$ is bivalent.*

Proof: For every i, $0 \leq i \leq n$, we prove that there is an execution α_i of at most $n+i$ processes which includes k complete rounds such that $C_{end}(\alpha_i)$ is bivalent and $|NY_{k+1}(\alpha_i)| < n - i + 1$. The proof is by induction on i.

Basis: $i = 0$. The execution $\alpha_0 = \alpha$ is an execution of n processes that includes k complete rounds; clearly, $C_{end}(\alpha_0)$ is bivalent and $|NY_{k+1}(\alpha_0)| < n + 1$.

Inductive step: Suppose the lemma holds for i, $0 \leq i \leq n-1$. That is, there is an execution α_i of $n+i$ processes with k complete rounds such that $C_{end}(\alpha_i)$ is bivalent and $|NY_{k+1}(\alpha_i)| \leq n - i$. If $|NY_{k+1}(\alpha_i)| < n - i$ then take $\alpha_{i+1} = \alpha_i$. Otherwise, $NY_{k+1}(\alpha_i)$ is not empty, and therefore, α_i includes exactly k complete rounds. By Lemma 6.7, there is an execution α_{i+1} of $n+i+1$ processes that includes k complete rounds, such that $C_{end}(\alpha_{i+1})$ is bivalent and $|NY_{k+1}(\alpha_{i+1})| < |NY_{k+1}(\alpha_i)| \leq n - i$, as needed. ■

Lemma 6.9 *For every k, $0 \leq k \leq \log N - 1$, there is an execution β_k of 2^{k+1} processes with k rounds such that $C_{end}(\beta_k)$ is bivalent.*

Proof: The proof is by induction on k.

Basis: $k = 0$. By Lemma 6.1, there is a bivalent initial configuration that contains two processes; β_0 is the empty execution of these two processes.

Inductive step: Suppose the lemma holds for k, $0 \leq k \leq \log N - 2$. That is, there is an execution β_k of at most 2^{k+1} processes with k rounds, such that $C_{end}(\beta_k)$ is bivalent. Since $2^{k+1} < N/2$, Lemma 6.8 implies the existence of an execution β_{k+1} of at most $2 \cdot 2^{k+1} = 2^{k+2}$ processes with $k+1$ rounds, such that $C_{end}(\beta_{k+1})$ is bivalent. ■

Applying Lemma 6.9 with $k = \log N - 1$, we get our lower bound:

Theorem 6.10 *In the totally anonymous model, the round complexity of a protocol solving binary consensus among N processes is $\Omega(\log N)$.*

7 Conclusions

This work studies the *totally anonymous shared memory* model of asynchronous distributed computing, where the processes are nonfaulty and the shared memory consists of read/write initialized registers only. It shows characterizations of the functions and the relations which are computable in this model. In particular, the most natural example of relation, the (fault-free) consensus task, can be computed in the model. We also study the complexity of solving consensus, and prove an $\Omega(\log N)$ lower bound on the number of registers and $\Omega(\log N)$ lower bound on the number of rounds needed for solving consensus.

The totally anonymous shared memory model has interesting relations to other models: it is more powerful than the fault-tolerant model [1, 10, 13], since consensus can be computed. It is also more powerful than the totally anonymous non-initialized model of Jayanti and Toueg [11], but is less powerful than their partially anonymous model.

Several issues remain open in this research, one of them in the gap between the protocol for consensus presented in this work that uses a linear number of registers and requires a linear number of rounds and the logarithmic lower bounds presented here. Other possible directions for future research are to consider stronger models, e.g., allowing partial synchronism or using randomized protocols.

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, Sept. 1993.

[2] D. Angluin. Local and global properties in networks of processors. In *Proceedings of the 12th ACM Symposium on Theory of Computing*, pages 82–93, 1980.

[3] H. Attiya, M. Snir, and M. Warmuth. Computing on an anonymous ring. *Journal of the ACM*, 35(4):845–876, Oct. 1988.

[4] H. Brit and S. Moran. Wait-freedom vs. bounded wait-freedom in public data structures. *Universal Journal of Computer Science*, pages 2–19, Jan. 1996.

[5] J. E. Burns and N. A. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, Dec. 1993.

[6] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, Jan. 1987.

[7] F. Fich, M. Herlihy, and N. Shavit. On the space complexity of randomized synchronization. *Journal of the ACM*, 45(5):843–862, Sept. 1998.

[8] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

[9] M. J. Fischer, S. Moran, S. Rudich, and G. Taubenfeld. The wakeup problem. *SIAM J. Comput.*, 25(6):1332–1357, Dec. 1996.

[10] M. P. Herlihy. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.*, 13(1):124–149, Jan. 1991.

[11] P. Jayanti and S. Toueg. Wakeup under read/write atomicity. In *Proceedings of the 4th International Workshop on Distributed Algorithms*, pages 277–288, 1990.

[12] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Prog. Lang. Syst.*, 4(3):382–401, July 1982.

[13] M. C. Loui and H. H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. In *Advances in Computing Research, Vol. 4*, pages 163–183. JAI Press, Inc., 1987.

[14] R. Lubitch and S. Moran. Closed schedulers: A novel technique for analyzing distributed protocols. *Distributed Computing*, 8(4):203–210, 1995.

[15] S. Moran and G. Taubenfeld. A lower bound on wait-free counting. *Journal of Algorithms*, 24:1–19, 1997.

[16] S. Moran and Y. Wolfsthal. An extended impossibility result for asynchronous complete networks. *Inf. Process. Lett.*, 26:141–151, 1987.

[17] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, Apr. 1980.

[18] E. Styer and G. L. Peterson. Tight bounds for shared memory symmetric mutual exclusion problems. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 177–192, 1989.

[19] G. Taubenfeld and S. Moran. Possibility and impossibility results in a shared memory environment. *Acta Inf.*, 33(1):1–20, 1996.