THE WAKEUP PROBLEM*

MICHAEL J. FISCHER†, SHLOMO MORAN‡, STEVEN RUDICH§, AND GADI TAUBENFELD¶

Abstract. We study a new problem, the *wakeup problem*, that seems to be fundamental in distributed computing. We present efficient solutions to the problem and show how these solutions can be used to solve the consensus problem, the leader election problem, and other related problems. The main question we try to answer is, how much memory is needed to solve the wakeup problem? We assume a model that captures important properties of real systems that have been largely ignored by previous work on cooperative problems.

1. Introduction.

1.1. The Wakeup Problem. The *wakeup problem* is a deceptively simple new problem that seems to be fundamental in distributed computing. The goal is to design a t-resilient protocol for n asynchronous processes in a shared memory environment such that at least p processes eventually learn that at least τ processes have waked up and begun participating in the protocol. Put another way, the wakeup problem with parameters n, t, τ and p is to find a protocol such that in any fair run of n processes with at most t failures, at least p processes eventually *know* that at least τ processes have taken at least one step in the past. 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.

In the wakeup problem, it is known a priori by all processes that at least $n - t$ processes will eventually wake up. The goal is simply to have a point in time at which the fact that at least τ processes have already waked up is *known* to p processes. It is not required that this time be the earliest possible, and faulty processes are included in the counts of processes that have waked up and that know about that fact. Note that in a solution to the wakeup problem, at least $p-t$ correct processes eventually learn that at least $\tau - t$ correct processes are awake and participating in the protocol.

The significance of this problem is two-fold. First, it seems generally useful to have a protocol such that after a crash of the network or after a malicious attack, the remaining correct processes can figure out if sufficiently many other processes remain active to carry out a given task. Second, a solution to this problem is a useful building block for solving other important problems such as the consensus [Abr88, Fis83, PSL80], leader election [FL87, Pet82], memory initialization [Hem89], phase synchronization [Mis91], and processor identity [LP90] problems.

1.2. A New Model. Much work to date on fault-tolerant parallel and distributed systems has been generous of the class of faults considered but rather strict in the requirements on the system itself. Problems are usually studied in an underlying model that is fully synchronous, provides each process with a unique name that is known to all other processes, and is initialized to a known state at time zero. We argue that none of these assumptions is realistic in today's computer networks, and achieving them even within a single parallel computer is becoming increasingly difficult and costly. Large systems do not run off of a single clock and hence are not synchronous. Providing processes with unique id's is costly and difficult and greatly complicates reconfiguring the system. Finally, simultaneously resetting all of the computers and communication channels in a large network to a known initial state is virtually impossible and would rarely be done even if it were possible because of the large destructive effects it would have on ongoing activities.

Our new model of computation makes none of these assumptions. It consists of a fully asynchronous collection of n identical anonymous processes that communicate via a *single* finite sized

* A PRELIMINARY VERSION OF THIS WORK APPEARED IN THE *PROCEEDINGS OF THE TWENTY-SECOND ANNUAL SYMPOSIUM ON THEORY OF COMPUTING (STOC),* BALTIMORE, MARYLAND, MAY 1990.

† Computer Science Department, Yale University, New Haven, CT 06520.
‡ Computer Science Department, Technion, Haifa 32000, Israel.
§ Computer Science Department Carnegie Mellon University, Pittsburgh, PA 15213.
¶ AT&T Bell Laboratories and the Open University of Israel.

This work was supported in part by ONR contract N00014-89-J-1980, by the National Science Foundation under grant CCR-8405478, by the Hebrew Technical Institute scholarship, by the Technion V.P.R. Funds - Wellner Research Fund, and by the Foundation for Research in Electronics, Computers and Communications, administrated by the Israel Academy of Sciences and Humanities.

shared register which is initially in an arbitrary unknown state. Access to the shared register is via atomic "read-modify-write" instructions which, in a single indivisible step, read the value in the register and then write a new value that can depend on the value just read. (When only atomic read and atomic write instructions are assumed the wakeup problem can not be solved even when $t = 0, \tau = 2$ and $p = 1$, since no process can ever learn that the others are awake if the processes are scheduled in a round robin fashion.)

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 we show how to solve decision problems such as the wakeup, consensus and leader election problems, in which a process makes an irrevocable decision after a finite number of steps.

Before proceeding, we should address two possible criticisms of shared memory models in general and our model in particular. First, most computers implement only reads and writes to memory, so why do we consider atomic read-modify-write instructions? One answer is that large parallel systems access shared memory through a communication network which may well possess independent processing power that enables it to implement more powerful primitives than just simple reads and writes. Indeed, such machines have been seriously proposed [GGK+84, Pea85]. Another answer is that part of our interest is in exploring the boundary between what can and cannot be done, and a proof of impossibility for a machine with read-modify-write access to memory shows *a fortiori* the corresponding impossibility for the weaker read/write model.

A second possible criticism is that real distributed systems are built around the message-passing paradigm and that shared memory models are unrealistic for large systems. Again we have several possible answers. First, the premise may not be correct. Experience is showing that message-passing systems are difficult to program, so increasing attention is being paid to implementing shared memory models, either in hardware (e.g. the Fluent machine [RBJ88]) or in software (e.g. the Linda system [CG89]). Second, message-passing systems are themselves an abstraction that may not accurately reflect the realities of the underlying hardware. For example, message-passing systems typically assume infinite buffers for incoming messages, yet nothing is infinite in a real system, and indeed overflow of the message buffer is one kind of fault to which real systems are subject. It is difficult to see how to study a kind of fault which is assumed away by the model. Finally, at the lowest level, communication hardware looks very much like shared memory. For example, a wire from one process to another can be thought of as a binary shared register which the first process can write (by injecting a voltage) and the second process can read (by sensing the voltage).

1.3. Space Complexity Results. The main question we try to answer is, how many values v for the shared register are necessary and sufficient to solve the wakeup problem? The answer both gives a measure of the communication-space complexity of the problem and also provides a way of assessing the cost of achieving reliability. We give a brief overview of our results below.

1.3.1. Fault-Free Solutions. First we examine what can be done in the absence of faults (i.e., $t = 0$). We present a solution to the wakeup problem in which one process learns that all other processes are awake (i.e., $p = 1$ and $\tau = n$), and it uses a single 4-valued register (i.e., $v = 4$). The protocol for achieving this is quite subtle and surprising. It can also be modified to solve the leader election problem. Based on this protocol, we construct a fault-free protocol that reaches consensus on one out of k possible values using a 5-valued register. Finally, we show that there is no fault-free solution to the wakeup problem with only two values (i.e., one bit) when $\tau \geq 3$.

1.3.2. Fault-Tolerant Solutions: Upper Bounds. We start by showing that the fault-free solution which uses a single 4-valued register, mentioned in the previous section, can actually tolerate t failures for any $\tau \leq ((2n-2)/(2t+1)+1)/2$. Using many copies of this protocol, we construct a protocol with $v = 8^{t+1}$ that tolerates t faults when $\tau \leq n-t$. Thus, if t is a constant, then a constant sized shared memory is sufficient, independent of n. However, the constant grows exponentially with t. An easy protocol exists with $v = n$ that works for any t and $\tau \leq n-t$. This means that the above exponential result is only of interest for $t \ll \log n$. Finally, we show that for any $t < n/2$, there is a t-resilient solution to the wakeup problem for any $\tau \leq \lfloor n/2 \rfloor + 1$, using a single $O(t)$-valued register.

1.3.3. Fault-Tolerant Solutions: A Lower Bound. We prove that for any protocol P that solves the wakeup problem for parameters n, t and τ, where $1 < t \leq 2n/3$ and $\tau > \lceil n/3 \rceil$, and for every $0 < \varepsilon \leq 1/2$, the number of shared memory values used by P is at least $(W+1)^{\alpha}$, where $W = \varepsilon t^2/(2(n-t))$ and $\alpha = 1/(\log_2((n-t)/((1-\varepsilon)t)+2))$. The proof is quite intricate and involves showing for any protocol with too few memory values that there is a run in which $n-t$ processes wake up and do not fail, yet no process can distinguish that run from another in which fewer than τ wake up; hence, no process knows that τ are awake.

When we take t to be a constant fraction of n we get the following immediate corollary: Let P be a protocol that solves the wakeup problem for parameters n, t and τ, where $t \geq n/c$ and $\tau > \lceil n/3 \rceil$. Let V be the set of shared memory values used by P. Let $\gamma = 1/(\log_2(c+1))$ and $\delta > 0$. Then, $|V| = \Omega(n^{\gamma-\delta})$. The corollary gives the bound we obtain in the case that $t = \Omega(n)$. However, when $t = O(n^{\xi})$ for $\xi < 1$ a constant, one gets that our lower bound $(W+1)^{\alpha} = O(1)$ and hence is not interesting.

1.4. Relation to Other Problems. We establish connections between the wakeup problem and two fundamental problems in distributed computing: the consensus problem and the leader election problem. These two problems lie at the core of many problems for fault-tolerant distributed applications [Abr88, AG85, CR79, DDS87, DKR82, DLS88, Fis83, FL87, FLM86, FLP85, HS80, KKM, KMZ84, Pet82, PKR84, PSL80, TKM89a, Tau91].

We show that: (1) any protocol that uses v values and solves the wakeup problem for $t < n/2$, $\tau > n/2$ and $p = 1$ can be transformed into t-resilient consensus and leader election protocols which use $8v$ values; and (2) any t-resilient consensus or leader election protocol that uses v values can be transformed into a t-resilient protocol which uses $4v$ values and solves the wakeup problem for any $\tau \leq \lfloor n/2 \rfloor + 1$ and $p = 1$.

Using the first result above, we can construct efficient solutions to both the consensus and leader election problems from solutions for the wakeup problem. The second result implies that the lower bound proved for the wakeup problem holds for these other two problems. As a consequence, the consensus and the leader election problems are space-equivalent in our model.

2. Definitions and Notations.

2.1. Protocols and Knowledge. An n-process *protocol* $P = (C, N, R)$ consists of a nonempty set C of *runs*, an n-tuple $N = (q_1, \ldots, q_n)$ of *process id's* (or *processes*, for short), and an n-tuple $R = (R_1, \ldots, R_n)$ of sets of *registers*. Informally, R_i includes all the registers that process q_i can access. We assume throughout this paper that $n \geq 2$.

A *run* is a pair (f, S) where f is a function which assigns initial values to the registers in $R_1 \cup \ldots \cup R_n$ and S is a finite or infinite sequence of events. (When S is finite, we also say that the run is finite.) An *event* $e = (q_i, v, r, v')$ means that process q_i, in one atomic step, first reads a value v from register r and then writes a value v' into register r. We say that the event e *involves* process q_i and register r, and that process q_i performs a *read-modify-write* operation on register r.

The *value* of a register at a finite run is the last value that was written into that register, or its initial value if no process wrote into the register. We use $value(r, \rho)$ to denote the value of r at a finite run ρ.

A register r is said to be *local* if there exists an i such that $r \in R_i$ and for any $j \neq i, r \notin R_j$. A register is *shared* if it is not local. In this paper we restrict attention to protocols which have exactly one register which is shared by all the processes (i.e., $|R_1 \cap \ldots \cap R_n| = 1$) and all other registers are local. We assume that all local registers of process q_i $(1 \leq i \leq n)$, have names of the form $r.i$. Furthermore, we assume that for any two processes q_i and q_j, the (local) register $r.i$ exists iff the register $r.j$ exists.

If S' is a prefix of S then the run (f, S') is *a prefix* of (f, S), and (f, S) is *an extension* of (f, S'). Let $\langle S; S' \rangle$ be the sequence obtained by concatenating the sequences S and S'. For a run $\rho = (f, S)$, let $\langle \rho; S' \rangle$ be an abbreviation for $(f, \langle S; S' \rangle)$. For any sequence S, let S_i be the subsequence of S containing all events in S which involve q_i. Runs (f, S) and (f', S') are *equivalent with respect to* q_i, denoted by $(f, S) \stackrel{i}{\sim} (f', S')$, iff $S_i = S'_i$. Let *null* denotes the empty sequence.

The set of runs of each protocol considered in this paper is assumed to satisfy the following five properties.

- ρ is a run iff every prefix of ρ is a run.
- Let ρ and ρ' be finite runs such that $\rho \stackrel{i}{\sim} \rho'$ and $value(r, \rho) = value(r, \rho')$. Then, $\langle \rho; (q_i, v, r, v') \rangle$ is a run iff $\langle \rho'; (q_i, v, r, v') \rangle$ is a run. That is, if some event can happen at a process q_i at

some point in a run, then the same event can happen at any run that is equivalent to that run w.r.t q_i provided that the register q_i access in that event has the same value in both run.

- Let $\langle \rho; (q_i, v, r, v')\rangle$ be a run. Then, $v = value(r, \rho)$. That is, it is possible to read only the last value that is written into a register.
- Let r be the single shared register. For any run ρ, there exists a run $(g, null)$ where $g(r) = value(r, \rho)$. That is, nothing can be assumed about the initial values.
- Let π be a permutation of $\{1, ..., n\}$, let S_π be the sequence of events S where for every $1 \leq i \leq n$ every appearance of q_i in S is replaced by $q_{\pi(i)}$, and let f_π be a function where $f_\pi(r) = f(r)$ for the shared register r, and $f_\pi(r.\pi(i)) = f(r)$ for any local register $r.i$. Then, if (f, S) is a run then (f_π, S_π) is also a run for every permutation π. That is, the processes are anonymous and identically programmed.

Notice that the above properties allow nondeterministic processes. However, for convenience, we will assume that processes are deterministic.

We are now ready to define the notion of knowledge in a shared memory environment. In the following, we use *predicate* to mean a set of runs.

Definition: For a process q_i, predicate b and finite run ρ, *process q_i knows b at ρ* iff for all ρ' such that $\rho \overset{i}{\sim} \rho'$, it is the case that $\rho' \in b$.

We say that a process p *learns* a predicate b in a run ρ if p knows b in ρ but it does not know b in any proper prefix of ρ.

For simplicity, we assume that a process always takes a step whenever it is scheduled. A process that takes infinitely many steps in a run is said to be *correct* in that run; otherwise it is *faulty.* We say that an infinite run is *l-fair* iff at least l processes are correct in it.

2.2. Wakeup, Consensus and Leader Election Protocols. In this subsection we formally define the notions of t-resilient wakeup, consensus and leader election protocols ($0 \leq t \leq n$). We say that a process q_i is *awake* in a run if the run contains an event that involves q_i. The predicate "*at least τ processes are awake*" is the set of all runs for which there exist τ different processes which are awake in the run. Note that a process that fails after taking a step is nevertheless considered to be awake in the run.

- A *wakeup protocol* with parameters n, t, τ and p is a protocol for n processes such that, for any $(n-t)$-fair run ρ, there exists a finite prefix of ρ in which at least p processes *know* that at least τ processes are awake in ρ.
  It is easy to see that a wakeup protocol exists only if $\max(p, \tau) \leq n - t$, and hence, from now on, we assume that this is always the case. We also assume that $\min(p, \tau) \geq 1$.
  In the following, whenever we speak about a solution to the wakeup problem without mentioning p, we are assuming that $p = 1$.
- A *t-resilient k-consensus protocol* is a protocol for n processes, where each process has a local read-only input register and a local write-once output register. For any $(n-t)$-fair run there exists a finite prefix in which all the correct processes decide on some value from a set of size k (i.e., each correct process writes a *decision value* into its local output register), the decision values written by all processes are the same, and the decision value is equal to the input value of some process.
  In the following, whenever we say "consensus" (without mentioning specific k) we mean "binary consensus", where the possible decision values are 0 and 1.
- Let P be a protocol for n processes, where each process has a local write-once output register, and let ρ be a finite run of P. We say that a process q_i *commits* to a value $v \in \{0, 1\}$ in ρ if q_i either has already written or eventually writes v to its output register in any $(n-t)$-fair extension of ρ in which q_i is correct. A process q_i is *elected* in ρ if q_i *knows* that it is committed to 1 in ρ. P is said to be a *t-resilient leader election protocol* if in any $(n-t)$-fair run of P, there exists a finite prefix in which exactly one process is elected and all other processes (correct or faulty) commit to the value 0. The elected process is called the *leader.* Notice, that here we need to use the notions "commits" and "elected" rather than "decides", since the elected leader might fail just before it writes to its output register (at which point it knows that it is committed to 1). Also, we observe that because processes are identical and anonymous, there can not be an a priori leader (if one process is elected without taking

any steps, then all processes do so). Thus, a process is elected in a run only if it participates in this run - a fact which is used in the sequel.

3. Fault-free Solutions. In this section, we develop the *See-Saw protocol*, which solves the fault-free wakeup problem using a single 4-valued shared register. Then we show how the See-Saw protocol can be used to solve the k-valued consensus problem. Finally, we show that it is impossible to solve the wakeup problem using only one shared bit.

To understand the See-Saw protocol, the reader should imagine a playground with a See-Saw in it. The processes will play the protocol on the See-Saw, adhering to strict rules. When each process enters the playground (wakes up), it sits on the up-side of the See-Saw causing it to swing to the ground. Only a process on the ground (or down-side) can get off and when it does the See-Saw must swing to the opposite orientation. These rules enforce a balance invariant which says that the number of processes on each side of the See-Saw differs by at most one (the heavier side always being down).

Each process enters the playground with two tokens. The protocol will force the processes on the bottom of the See-Saw to give away tokens to the processes on the top of the See-Saw. Thus, token flow will change direction depending on the orientation of the See-Saw. Tokens can be neither created nor destroyed. The idea of the protocol is to cause tokens to concentrate in the hands of a single process. A process seeing $2k$ tokens knows that at least k processes are awake. Hence, if it is guaranteed that eventually some process will see at least 2τ tokens, the protocol is by definition a wakeup protocol with parameter τ, even if the process does not know the value of τ and hence does not know when the goal has been achieved.

Following is the complete description of the See-Saw protocol. The 4-valued shared register is easily interpreted as two bits which we call the "token bit" and the "See-Saw" bit. The two states of the token bit are called "token present" and "no token present". We think of a public *token slot* which either contains a token or is empty, according to the value of the token bit. The two states of the See-Saw bit are called "left side down" and "right side down". The "See-Saw" bit describes a virtual See-Saw which has a left and a right side. The bit indicates which side is down (implying that the opposite side is up).

Each process remembers in private memory the number of tokens it currently possesses and which of four states it is currently in with respect to the See-Saw: "never been on" "on left side", "on right side", and "got off". A process is said to be on the up-side of the See-Saw if it is currently "on left side" and the See-Saw bit is in state "right side down", or it is currently "on right side" and the See-Saw bit is in state "left side down". A process initially possesses two tokens and is in state "never been on".

We define the protocol by a list of rules. When a process is scheduled, it looks at the shared register and at its own internal state and carries out the first applicable rule, if any. If no rule is applicable, it takes a null step which leaves its internal state and the value in the shared register unchanged.

Rule 1: *(Start of protocol)* Applicable if the scheduled process is in state "never been on". The process gets on the up-side of the See-Saw and then flips the See-Saw bit. By "get on", we mean that the process changes its state to "on left side" or "on right side" according to whichever side is up. Since flipping the See-Saw bit causes that side to go down, the process ends up on the down-side of the See-Saw.

Rule 2: *(Emitter)* Applicable if the scheduled process is on the down-side of the See-Saw, has one or more tokens, and the token slot is empty. The process flips the token bit (to indicate that a token is present) and decrements by one the count of tokens it possesses. If its token count thereby becomes zero, the process flips the See-Saw bit and gets off the See-Saw by setting its state to "got off".

Rule 3: *(Absorber)* Applicable if the scheduled process is on the up-side of the See-Saw and a token is present in the token slot. The process flips the token bit (to indicate that a token is no longer present) and increments by one the count of tokens it possesses.

Note that if a scheduled process is on the down-side, has $2k-1$ tokens, and a token is present in the token slot, then, although no rule is applicable, the process nevertheless sees a total of $2k$ tokens and hence knows that k processes have waked up.

The two main ideas behind the protocol can be stated as invariants.

TOKEN INVARIANT: The number of tokens in the system is either $2n$ or $2n+1$ and does not change at any time during the protocol. (The number of tokens in the system is the total number of tokens possessed by all of the processes, plus 1 if a token is present in the token bit slot.)

Proof: The number of tokens in the starting configuration is 2n with the possible addition of one token present in the token bit slot. The rules that effect tokens are rules 2 and 3 both of which maintain the token invariant.

BALANCE INVARIANT: The number of processes on the left and right sides of the See-Saw is either perfectly balanced or favors the down-side of the See-Saw by one process.

Proof: The see-saw starts empty, zero on either side. Rule 1 preserves the invariant because a process gets on the up side and then flips the see-saw. If a process runs out of tokens, it must be on the down side of the see-saw; hence, when rule 2 is applied the invariant is maintained.

Theorem 3.1: *Let $t = 0$. The See-Saw protocol uses a 4-valued shared register and is a wakeup protocol for n, t, τ (and $p = 1$), where n and τ are arbitrary and $t = 0$. (Note that the rules for the protocol do not mention n or τ.)*

Proof: By token invariant, there are no more than $2n+1$ tokens in the system. At most two come from each player; at most one comes from the initialized state of the token bit. Hence if a process sees 2τ tokens, it has to be the case that at least τ processes are awake.

Next, we argue that the protocol comes to a state where everybody has awaken and there is only one process remaining on the see-saw. We know there will be a time when everybody is awake. Furthermore, for any number of processes $m \geq 2$ still active on the see-saw, there will be a future time when there are only $m-1$ processes on the see-saw: By the balance invariant, there are some processes on both sides and hence eventually either rule 2 or rule 3 are applicable (i.e., there is no deadlock). Each process has awaken up, hence, rule 1 will no longer apply. Applying rules 2 and 3 will cause tokens to flow from the down side to the up side; eventually the token count of a down side process will become zero and the process will get off the See-Saw. Hence eventually there will be only one process remaining on the see-saw. This process will see $2n$ tokens and will know that all other processes are awake. $\square$

In applications of wakeup protocols, it is often desirable for the processes to know the value of τ so that a process learning that τ processes are awake can stop participating in the wakeup protocol and take some action based on that knowledge. The See-Saw protocol can be easily modified to have this property by adding a termination rule immediately after Rule 1:

Rule 1a: *(End of protocol)* Applicable if the scheduled process is on the See-Saw and sees at least 2τ tokens, where the number of tokens the process sees is the number it possesses, plus one if a token is present in the token slot. The process thus knows that τ processes have waked up. It gets off the See-Saw (i.e., terminates) by setting its state to "got off".

The See-Saw protocol can also be used to solve the leader election problem by electing the first process that sees $2n$ tokens. By adding a $5th$ value, everyone can be informed that the leader was elected, and the leader can know that everyone knows. Now, the leader can transmit an arbitrary message, for example a consensus value, to all the other processes without using any more new values through a kind of serial protocol. This leads to our next theorem.

Theorem 3.2: *In the absence of faults, it is possible to reach consensus on one of k values using a single 5-valued shared register.*

Proof: Assume the processes have been running the See-Saw protocol in which each process initially has 2 tokens. A process becomes leader when it accumulates $2n$ tokens, at which time possibly one more token remains elsewhere in the system.

Let *end* be a $5th$ value. The leader now puts *end* in the shared register. Any process seeing *end* for the first time replaces it with (*no token present, left side down*). The leader repeats this $n-1$ times, waiting each time for *end* to be removed from the register, after which time each other process knows of the existence of the leader. When the leader notices the last *end* disappear, it knows that everyone knows.

Note that at the start of these exchanges, all other processes save one are out of tokens and will ignore all messages except *end*. Any non-leader process possessing a single token will either ignore the message (*no token present, left side down*), or it will change it to (*token present, right side down*), depending on its type bit, and thereafter ignore all messages except *end*. Thus, from the time the leader is elected until the time that everyone knows of its election, the only possible shared register values are *end*, (*no token present, left side down*), and (*token present, right side down*). Hence, the remaining two values, (*no token present, right side down*), and (*token present, left side down*), can be reused to initiate sending the message, for they will not appear until after the leader knows that everyone knows. Call these values $data1$ and $ack1$, respectively. Call values (*no token present, left side down*) and (*token present, right side down*) $data2$ and $ack2$, respectively.

Let $1 \leq m \leq k$ be the consensus value, which we take to be the leader's initial value. Here is the protocol that the leader now uses to send m to all other processes. The leader executes m data phases. Each process counts the number of data phases executed. At the end of the m phases, the leader terminates the protocol by putting *end* back in the register. Each process terminates when it sees *end*, in which case it also knows the number of phases and hence the consensus value.

The first data phase involves the leader putting $data1$ into memory $n-1$ times. Each follower process, upon seeing $data1$, replaces it with $ack1$, increments its phase counter, and enters the next phase, where it waits for $data2$ or *end*. The second phase uses $data2$ and $ack2$, and subsequent phases alternate between the two versions of the values, odd numbered phases using $data1$ and $ack1$, and even numbered phases using $data2$ and $ack2$. □

Finally, we claim that the See-Saw protocol cannot be improved to use only a single binary register. A slightly weaker result than Theorem 3.3 was also proved by Joe Halpern [Hal]. The question whether 3 values suffice is still open.

Theorem 3.3: *There does not exist a solution to the wakeup problem which uses only a single binary register when* $\tau \geq 3$.

In order to prove Theorem 3.3, we first prove a simple lemma. We say that a process writes the value a_∞, if the process writes a, and at the infinite extension in which this process is the only one that is activated, a appears infinitely many times. We notice that when the memory is bounded, for any run ρ and any process p, if p is run alone from ρ then p must eventually write a_∞, for some a.

Lemma 3.1: *In any wakeup protocol where* $\tau \geq 2$, *if the initial value is* a *and only one process wakes up and it is activated alone forever then it will never write* a_∞.

Proof: Assume to the contrary that the lemma does not hold. We show that this leads to a contradiction by constructing a n-fair run in which the initial value of that shared register is a, the value a appears infinitely many times, and yet no process knows that any other process is awake. We construct the run ρ by activating the processes in a round robin fashion infinitely many times, starting with a as the initial value. Each time a process is scheduled, we let it run until it writes a_∞. Each process cannot distinguish ρ from the run, constructed similarly, in which it is the only process that is activated. Hence no process ever knows that any other process is awake. □

Proof of Theorem 3.3: We first assume that n is even, and construct a n-fair run, called ρ, such that in each prefix of that run each process only knows that one other process is awake.

Assume that the initial value is $b \in \{0, 1\}$, let q be an arbitrary process, and consider the following scenario. First q runs alone until it writes a_∞ (by Lemma 3.1 $a \neq b$). At that point we interfere and flip the shared bit so that its value is again b. Afterwards we let q continue until it writes a_∞ again and then we flip the bit and so on. Let $flip(b)$ be the number of times q writes a_∞ at such an infinite run. We consider the two possible cases:

The first case is when both $flip(0)$ and $flip(1)$ are infinite. We construct the run ρ by activating the processes in a round robin fashion infinitely many times, starting with 0 as the initial value. Each time a process is scheduled, if the value of the shared bit is a (b) we let it run until it writes b_∞ (a_∞). Each process cannot distinguish ρ from the run, constructed similarly, in which only two processes participate. Hence no process ever knows that more than one other process is awake.

The other case is the negation of the previous one. Assume w.l.o.g. that $flip(0) = k$ for some positive number k, and that $flip(0) \leq flip(1)$. We construct the run ρ by first activating the processes in a round robin fashion exactly as in the previous construction but only for k rounds, starting with 0 as the initial value. After k rounds the value of the shared bit is 0. We extend this run to a n-fair run by continuing activating the processes in a round robin fashion letting each process make one or more steps whenever it is scheduled until it writes 0_∞ (note that this is always possible since $flip(0) \leq flip(1)$). As in the previous case, no process can distinguish this run from the run, constructed similarly, in which only two processes participate. Hence no process ever knows that more than one other process is awake. This completes the proof when n is even.

Assume that n is odd. Let $m = n - 1$. Since m is even we can construct exactly as before a m-fair run, called ρ, in which no process ever knows that more than one other process is awake. Let q be the remaining process. We now construct ρ' as follows. We start with 1 as the initial value and let q run until it writes (as is assured by Lemma 3.1) 0_∞. Then alternately we let the other m processes run as in ρ until 0 appears, then we let q take one or more steps until 0 appears (this is assured since previously q wrote 0_∞) and so on. Clearly, q cannot distinguish ρ' from a run where it is the only process that is activated, and hence never know that any of the other processes is awake. The other processes, cannot distinguish ρ' from ρ and hence never know (as in ρ) that more than one other process is awake. $\Box$

4. Fault-tolerant Solutions. In this section, we explore solutions to the wakeup problem which can tolerate $t > 0$ process failures. The See-Saw protocol, presented in the previous section, cannot tolerate even a single crash failure for any $\tau > n/3$. The reason is that the faulty process may fail after accumulating $2n/3$ tokens, trapping two other processes on one side of the See-Saw, each with $2n/3$ tokens. When $\tau \leq n/3$, the See-Saw protocol can tolerate at least one failure. As the parameter τ decreases, the number of failures that the protocol can tolerate increases. This fact is captured by the following theorem.

Theorem 4.1: *The See-Saw protocol is a wakeup protocol for n, t, τ, where*

$$\tau \leq \frac{(2n-1)/(2t+1)+1}{2}.$$

Proof: Failures affect the protocol in two ways: First, tokens possessed by a failed process are lost to the system. Second, failures can disrupt the balance condition on the number of active processes of each type. Thus, after t failures, up to $t(2\tau - 1)$ tokens can be lost, and the number of active processes of each type can differ by up to $t+1$. (If a faulty process accumulates 2τ tokens, it knows that at least τ processes are awake, and the goal of the protocol is achieved.) This implies that when one reaches a stage in which there are either no emitters or no absorbers, there can remain as many as $t+1$ active processes. In order to guarantee termination, we must be assured that at least one of these remaining processes holds at least 2τ tokens. Since the other t active processes can each hold $2\tau - 1$ tokens, the total number of tokens remaining after t failures must be at least $2\tau + t(2\tau - 1)$. (Notice that at the point when one process accumulates 2τ tokens, there is no token in the shared register.) Hence, we must have $2n - t(2\tau - 1) \geq (t+1)(2\tau - 1) + 1$. Solving, we get $2n \geq (2t+1)(2\tau - 1) + 1$, so $\tau \leq ((2n-1)/(2t+1)+1)/2$. $\Box$

If one insists that some non-failing process learns that τ non-failing processes have waked up, then a process terminates when it collects $2(\tau + t)$ tokens, and each failing process can take at most $2(\tau + t)$ tokens with it (since it stops accumulating tokens when it has that number). Hence, we get the inequality $2n - t(2(\tau + t)) \geq (t+1)(2(\tau+t) - 1) + 1$. Solving, we get $2n \geq 2(2t+1)(\tau + t) - t$, so $\tau \leq (2n+t)/(2(2t+1)) - t$. We note that the See-Saw protocol can tolerate up to $n/2 - 1$ *initial failures* [FLP85, TKM89b].

As we can see the See-Saw protocol needs only 4 values but is very sensitive to failures. Let us define $\psi = \tau/(n-t)$ as the *sensitivity parameter* of a wakeup protocol. Clearly, in the See-Saw protocol, when t is a constant fraction of n, the limit of ψ as n goes to infinity is zero. In the rest of this section, we present three t-resilient wakeup protocols. In the first two protocols $\psi = 1$, but they need n and 8^{t+1} values. In the third protocol $\psi \geq 1/2$, but it needs only $O(t)$ values.

Theorem 4.2: *For any $t < n/6$, there is a wakeup protocol which uses a single 8^{t+1}-valued register and works for any $\tau \leq n - t$.*

Proof: The solution is constructed using $t+1$ copies of the See-Saw protocol. Before going into details let us first reexamine the See-Saw protocol. Consider the following situation. There are only three processes, and initially there is a token in the shared variable. Let each process make one move. Now there are two emitters and one absorber. If at that point the absorber fails the other two processes are captured forever in the protocol.

It is not difficult to see that t faulty processes can trap at most $t+1$ other processes in an execution of the See-Saw protocol. (I.e., if there is a deadlock then at most $t+1$ correct processes have not yet terminated.) The proof of that fact follows from the invariant that the difference between emitters and absorbers is at most one.

Also, we observe that if we have a leader which is guaranteed not to fail then one bit is sufficient in order for the leader to learn that $n-t$ processes are awake, assuming up to t failures. This goes as follows: When the leader reads 1 it writes 0 otherwise it waits. When a slave reads 0 it writes 1 otherwise it waits. Each slave changes the bit exactly two times. When the leader learns that the bit has been changed $2(n-t)-3$ times from 0 to 1, it knows that $n-t$ processes (including itself) are awake. Call this trivial protocol the *Leader protocol.*

Using these observations we are ready to present, for any $t < n/6$, a t-resilient wakeup protocol for $\tau = n - t$, which uses a single 8^{t+1}-valued register. In this protocol the processes participate in $t+1$ See-Saw protocols in a sequential manner. That is, processes get on the i'th protocol only after they get off the $i-1$ protocol. In addition all processes participate in $t+1$ Leader protocols in parallel. That is, each process participates in one See-Saw protocol and in $t+1$ Leader protocols at the same time.

Each process behaves according to the following rules,

For all $1 \leq i \leq t+1$:

- A process that accumulates $n+1$ tokens in the i'th See-Saw protocol becomes the *leader* of that protocol, and takes the role of the leader in the i'th Leader protocol, (and participate as a slave in all other Leader protocols).
- At any time a process that is not a leader at the i'th See-Saw protocol participates as a slave in the i'th Leader protocol.
- Once a leader is elected in the i'th See-Saw protocol, it immediately stops participating in this protocol and participates only in all the $t+1$ Leader protocols. (The justification for that is that, if the leader never fails then eventually it will learn that $n-t$ processes are awake. If it does fail then we can assume w.l.o.g. that it always fails immediately after it is elected.)

This completes the description of the protocol.

The correctness proof is as follows. Since, once a process accumulates $n+1$ tokens in the i'th See-Saw protocol, it stop participating in it, no other process will ever accumulate $n+1$ tokens at this See-Saw protocol. Hence, at each See-Saw protocol at most one leader is elected and at each Leader protocol at most one process participates as a leader. The next observation is that, if no reliable leader is elected in one of the first t See-Saw protocols, then eventually a reliable leader is elected at the $t+1$ See-Saw protocol. The reason for that is as follows. Assuming no reliable leader is elected in the first t See-Saw protocols implies that t processes already fail, and hence any process that participates in the $t+1$ protocol has to be reliable. The total number of processes that can either fail or be trapped in the first t See-Saw protocols is at most $3t$. Hence, since $t < n/6$, it follows that more than $n/2$ processes will eventually participate in the $t+1$ See-Saw protocol and one of them will eventually be elected. Thus, eventually a reliable leader is elected and it will learn that $n-t$ processes are awake by participating as a leader in one of the leader protocols. □

We notice that instead of using a single 8^{t+1}-valued shared register, it is possible to use $t+1$ 4-valued registers and $t+1$ binary registers where a process can read-modify-write only one such register at a time. Although the protocol sensitivity parameter is optimal, its space complexity grows exponentially with t. Notice that when the number of failures t is a constant, one process can learn that $n-t$ processes are awake with a constant number of values.

Theorem 4.3: *For any $t < n$, there is a wakeup protocol which uses a single n-valued register and works for any $\tau \leq n-t$.*

Proof: The solution uses a single register called *counter*, whose values are $\{0, ..., n-1\}$. Each process initially records the value of the counter and increments it by 1 (mod n). Thereafter, it reads the value of the counter until it finds out that the counter has advanced by at least τ, which implies that at least τ processes are awake. Clearly, at least one reliable process must see this. □.

For later reference we call the above protocol *the counter protocol.* Although the Counter protocol sensitivity parameter is optimal (i.e., $\psi = 1$) its space requirement seem to be to big, when t is small. Hence, the protocol of Theorem 4.2, is better than the Counter protocol for $t \ll log(n)$. In our next solution ψ is not optimal and ranges from 1/2 to 1 depending on the value of t, however its space complexity is linear in t.

Theorem 4.4: *For any $t < n/2$, there is a wakeup protocol which uses a single $O(t)$-valued register and works for any $\tau \leq \lfloor n/2 \rfloor + 1$.*

Proof: In the sequel, we describe for any $t < n/10$, a wakeup protocol for $\tau = \lfloor n/2 \rfloor + 1$, which uses a single $O(t)$-valued register. The protocol is presented together with its correctness proof. When $t > n/10$ we may use the Counter protocol mentioned before. The protocol is obtain by using a counter and the See-Saw protocol. The size of the counter is n/k where the value of k is defined later. Each process executes the See-Saw protocol and accesses the counter as follows.

- Each process starts with only one token.
- A process increments the counter if and only if the following holds,
  - it is an absorber (in the See-Saw protocol), and
  - it accumulates k tokens, and
  - if it is the second time the process tries to increment the counter then it has to be the case that the counter has to be incremented at least t times from the first time it has incremented the counter.
- When a process increments the counter it erases all the k tokens it holds, and continues to participate as an absorber in the See-Saw protocol.
- An absorber in the See-Saw protocol never collects more than k tokens. (If it has k token and can not increment the counter, then it does nothing until it either becomes an emitter or can increment the counter.)

Once a process learns that the counter has been incremented by more than $n/(2k)$ times, it knows that more than half of the processes are awake.

It remains to decide what is the value of k as a function of n and t. We consider the following observations.

1. If there is a deadlock then at most $4t + 1$ processes are trapped in it. ($2t$ by being faulty or absorbers with k tokens that do not fulfill the conditions to increment the counter, and $2t + 1$ emitters.) Since each process may hold at most k tokens $(4t + 1)k + 1$ tokens may be lost. (The 1 is for the token in the shared register.)
2. At the time the first non faulty process reads (and remembers) the counter value, the counter has been incremented at most t times from start up time of the protocol. Hence, the kt tokens that are used for these t increments may be lost.
3. From (1) and (2) at most $(5t + 1)k + 1$ tokens may be lost.
4. To ensure that eventually one correct process will read the counter and will learn that more than $n/2$ processes are awaken, it is enough to require that $(5t + 1)k + 1 < n/2$.
5. In order that the number of values of the counter will be $O(t)$ we should choose k such that $n/k = O(t)$.

Hence, we end with three requirements for k: (1) k is a positive integer,
(2) $k < (n - 2)/(10t + 2)$, and (3) $n/k = O(t)$. From (1) and (2) and the fact that t is an integer, it follows that $t < n/10$. Taking $k = \lfloor (n - 3)/(10t + 2) \rfloor$ is the best choice for k. For the See-Saw protocol we need 4 values, and for the Counter protocol we need n/k values, which (for large n) is less then $12t$. Hence, a single $48t$-valued register suffices for the protocol we just described. $\square$

In this last protocol, one process learns that $n/2$ other processes are awake. In order for one process to learn that ln processes are awake, we should replace "$(5t+1)k+1 < n/2$" in Observation 4, with "$(5t+1)k+1 < (1-l)n$" and get that a single $O(t/(1-l))$-valued register suffices for solving the wakeup problem for $\tau = ln$.

5. A Lower Bound. In this section, we establish a lower bound on the number of shared memory values needed to solve the wakeup problem, where *only one* process is required to learn that τ processes are awake, assuming t processes may crash fail (i.e., $p = 1$). To simply exposition, we assume that $1 < t \leq 2n/3$ and $\tau > \lceil n/3 \rceil$. Also, recall that we already assumed that $\tau \leq n - t$. For the rest of this section, let $0 < \varepsilon \leq 0.5$ be fixed (but arbitrary), and let

$$W = \frac{\varepsilon t^2}{2(n-t)}; \qquad \alpha = \frac{1}{\log_2(\frac{n-t}{(1-\varepsilon)t} + 2)}. \tag{1}$$

Theorem 5.1: *Let P be a protocol that solves the wakeup problem for parameters n, t and τ. Let V be the set of shared memory values used by P. Then $|V| > (W+1)^\alpha$.*

When we take t to be a constant fraction of n we get the following immediate corollary.

Corollary 5.1: *Let P be a protocol that solves the wakeup problem for parameters n, t and τ, where $t \geq n/c$. Let V be the set of shared memory values used by P. Let $\gamma = 1/(\log_2(c+1))$ and $\delta > 0$. Then, $|V| = \Omega(n^{\gamma-\delta})$.*

Theorem 5.1 is immediate if V is infinite, so we assume throughout this section that V is finite. The proof consists of several parts. First we define a sequence of directed graphs whose nodes are shared memory values in V. Each component C of each graph in the sequence has a cardinality k_C and a weight w_C. We establish by induction that $w_C < k_C^{1/\alpha} - 1$. Finally, we argue that in the last graph in the sequence, every component C has weight $w_C \geq W$. Hence, $|V| \geq k_C > (W+1)^\alpha$.

5.1. Reachability Graphs and Terminal Graphs. Let V be the alphabet of the shared register. We say that a value $a \in V$ appears m times in a given run if there are (at least) m different prefixes of that run where the value of the shared register is a.

$a \xrightarrow{r} b$ denotes that there exists a run in which *at most* r processes participate, the initial value of the shared register is a, and the value b appears at least once.

$a \overset{r}{\Longrightarrow} b$ denotes that there exists a run in which *exactly* r processes participate, each process that participates takes infinitely many steps, the initial value of the shared register is a, and the value b appears infinitely many times.

Clearly, $a \overset{r}{\Longrightarrow} b$ implies $a \xrightarrow{r} b$ but not vice versa. Also, for every a and for every $r \leq n$, there exists b such that $a \overset{r}{\Longrightarrow} b$.

We use the following graph-theoretic notions. A directed multigraph[1] G is *weakly connected* if the underlying undirected multigraph of G is connected. A multigraph $G'(V', E')$ is a *subgraph* of $G(V, E)$ if $E' \subseteq E$ and $V' \subseteq V$. A multigraph G' is a *component* of a multigraph G if it is a weakly connected subgraph of G and for any edge (a, b) in G, either both a and b are nodes of G' or both a and b are not in G'. A node is a *root* of a multigraph if there is a directed path from every other node in the multigraph to that node. A *rooted graph* (rooted component) is a graph (component) with at least one root.

A *labeled* multigraph is a multigraph together with a label function that assigns a *weight* in $\mathbf{N}$ to each edge of G. The *weight* of a labeled multigraph is the sum of the weights of its edges. We now define the notion of a reachability graph of a given protocol P.

Definition: Let V be the set of shared memory values of protocol P. The *reachability graph G of protocol P* is the labeled directed multigraph with node set V, and which has an edge from node a to node b labeled with r iff $a \overset{r}{\Longrightarrow} b$ holds. (Note that there may be several edges with different labels between the same two nodes. Note also that G is finite since $a \overset{r}{\Longrightarrow} b$ implies that $r \leq n$.)

Definition: A graph C is *closed at node a w.r.t. G* if a is in C and for every node b in G, if (a, b) is an edge of G then b is in C.

Definition: A multigraph T is *terminal w.r.t. G* if T is a subgraph of G, all of T's components are rooted, and T has a component C with root a among its minimal weight components that is closed at node a w.r.t. G.

In the rest of the section we prove Theorem 5.1 by constructing a multigraph T which is terminal w.r.t. G, in which every component with weight w and size k satisfies $k^{1/\alpha} - 1 > w \geq W$.

[1] A multigraph can have several edges from a to b.

5.2. Reachability Graphs. The reachability graphs are defined for all protocols. Now we concentrate on such graphs constructed from wakeup protocols only. We show that when the weight of a rooted subgraph, say C, is sufficiently small, an edge exists with a label q from a root of C to a node not in C and we can bound the size of q.

For later reference we call the set of the following three inequalities,

$$\begin{array}{ll} \text{(i)} & zq + (z-1)w \leq n, \\ \text{(ii)} & zq \geq n - t, \\ \text{(iii)} & \max(q, w) < \tau \end{array}$$

the *zigzag inequalities.* These inequalities play an important role in our exposition.

Lemma 5.1: *Let G be a reachability graph of a wakeup protocol with parameters n, t, τ, and let C be a rooted subgraph of G with root a and weight w. If there exist positive integers z and q that satisfy the zigzag inequalities, then there exists a node b of G such that $a \overset{q}{\Longrightarrow} b$, and every such node b is not in C.*

Proof: Let z and q be positive integers that satisfy the zigzag inequalities. By (i), $q \leq n$, so there exists b such that $a \overset{q}{\Longrightarrow} b$. We show that $b \notin C$.

Assume to the contrary that $b \in C$. Let ρ be a q-fair run starting from a in which b is written infinitely often. Since b is in C, there is a path from b to a such that the sum of all the labels of edges in that path is at most w and hence $b \xrightarrow{w} a$. This allows us to construct a run with zq non-faulty processes starting with a as follows:

> Run q processes according to ρ until b is written. Run w processes until a is written. (This must be possible since $b \xrightarrow{w} a$.) Let these w processes fail. Run a second group of q processes according to ρ until b is written. Run a second group of w processes until a is written, and let them fail. Repeat the above until the z^{th} group of q processes have just been run and b has again been written. At this point, zq processes belong to still-active groups, and $(z-1)w$ processes have died. If any processes remain, let them die now without taking any steps. Now, an infinite run ρ' on the active processes can be constructed by continuing to run the first group according to ρ until b is written again, then doing the same for the second through z^{th} groups, and repeating this cycle forever.

The result is a zq-fair run. Moreover, no reliable process can distinguish this run from ρ, and hence no reliable process ever knows (in ρ') that more than q processes are awake. Also, obviously, no faulty process knows that more than w processes are awake. Since $\max(q, w) < \tau$ and $zq \geq n-t \geq \tau$, this leads to a contradiction to the assumption that the protocol solves the wakeup problem. □

Lemma 5.2: *Assume $1 \leq w \leq W$, where w is an integer. Let*

$$q = \left\lceil \frac{w(n-t)}{(1-\varepsilon)t} \right\rceil, \qquad z = \left\lfloor \frac{n-t}{q} \right\rfloor + 1. \tag{2}$$

Then z and q are positive integers which satisfy the zigzag inequalities.

Proof: From the assumption that $1 \leq w$, $0 < \varepsilon \leq 0.5$ and $t \leq 2n/3$ it follows that $q \geq \lceil 1/(2(1-\varepsilon)) \rceil = 1$, and hence both z and q are positive integers.

To prove inequality (i), observe that from (2) it follows that $z - 1 \leq (n-t)/q$. Thus,

$$zq = (z-1)q + q \leq n - t + q. \tag{3}$$

Also, since $1 \leq w \leq W$ and $\varepsilon \leq 0.5$, it follows from (2) that

$$q \leq \left\lceil \frac{\varepsilon t}{2(1-\varepsilon)} \right\rceil \leq \lceil \varepsilon t \rceil. \tag{4}$$

Hence, from (3) and (4) it follow that,

$$zq \leq n - t + \lceil \varepsilon t \rceil = n - \lfloor t - \varepsilon t \rfloor. \tag{5}$$

Next we show that

$$(z-1)w \leq \lfloor t - \varepsilon t \rfloor. \tag{6}$$

Notice that,

$$q = \left\lceil \frac{w(n-t)}{(1-\varepsilon)t} \right\rceil \geq \frac{w(n-t)}{(1-\varepsilon)t}. \tag{7}$$

As already mentioned, from (2) it follows that $z - 1 \leq (n-t)/q$, hence by using (7) we get that $z - 1 \leq (1-\varepsilon)t/w$. Since both z and w are integers, $(z-1)w \leq \lfloor t - \varepsilon t \rfloor$. Thus, using (5) and (6) we get that inequality (i) is satisfied.

Inequality (ii) is satisfied immediately since by (2) $z > (n-t)/q$.

Finally, we show that inequality (iii) is satisfied. Recall that we assume that $t \leq 2n/3$ and $\tau > \lceil n/3 \rceil$. It follows from these assumptions that $\tau > \lceil t/2 \rceil$. Since $q \leq \lceil \varepsilon t \rceil \leq \lceil t/2 \rceil$, obviously $q < \tau$. Also, since $w \leq W$ and $t \leq 2n/3$, substituting in (1) gives $w \leq \varepsilon t \leq n/3$, and hence $w < \tau$. □

5.3. Main Construction. In this subsection, we first prove that any rooted component of any terminal graph w.r.t. G has *weight* $> W$. Then we use this to construct a subgraph T of G all of whose rooted components have size $> (W+1)^\alpha$.

Lemma 5.3: *Let G be the reachability graph of a wakeup protocol with parameters n, t, τ and let T be terminal w.r.t. G. Any rooted component of T has weight $> W$.*

Proof: Assume that the lemma is false. Then T has a minimal-weight component C with root a, and weight $0 \leq w \leq W$, such that C is closed at a. If $w = 0$ then $q = 1$ and $z = n - t$ satisfy the zigzag inequalities, otherwise by Lemma 5.2, there exist positive integers q and z that satisfy the zigzag inequalities. From Lemma 5.1, there is a node b not in C and an edge $a \stackrel{q}{\Longrightarrow} b$ in G, contradicting the assumption that C is closed at a. □

Lemma 5.4: *Let G be the reachability graph of a wakeup protocol with parameters n, t, τ. There exists a subgraph T of G, all of whose rooted components have size $> (W+1)^\alpha$.*

Proof: The following procedure constructs T by adding edges one at a time to an *initial* subgraph T_0 of G until step 2 fails. The *initial* subgraph T_0 consists of all the nodes of G. First, for each node a we place exactly one outgoing edge $a \stackrel{1}{\Longrightarrow} b$ in T_0, and then we delete from each cycle one arbitrary edge. We note two facts about T_0: (1) for every edge $a \stackrel{1}{\Longrightarrow} b$, $a \neq b$, and (2) every component of T_0 is a directed rooted tree. Fact (1) is a simple variant of Lemma 5.1, while (2) follows from the observation that every component of T_0 has k vertices and $k-1$ edges for some $k \geq 2$, the out degree of every vertex is at most one, and it contains no cycles.

At any stage of the construction, every component of the graph built so far will be a directed rooted tree. Added edges always start at a root and end at a node of a different component. After adding an edge (a, b), the components of a and b are joined together into a single component whose root is the root of b's component, and the weight of the new component is the sum of the weights of the two original components plus the label of the edge from a to b.

<u>**Procedure for adding a new edge to T:**</u>

Step 1: *Select an arbitrary component C of minimal weight w, with root a.*

Step 2: *If $w \leq W$ then find the smallest integer q for which there is an edge $a \stackrel{q}{\Longrightarrow} b$ in G such that b is not in C. This step fails if $w > W$.*

Step 3: *Place the edge $a \stackrel{q}{\Longrightarrow} b$ into T.*

First note that, by Lemma 5.3, if $w \leq W$ then T is not terminal, and hence Step 2 cannot fail. Let T_i be a graph that is constructed after i applications of the above procedure, where T_0 is an initial graph as defined above. Clearly, any such sequence $\{T_0, T_1, \ldots\}$ is finite. Let T_{last} be the last element in this sequence. Then the weight of any component in T_{last} is greater than W.

Let $\beta = 1/\alpha$. We prove by induction on i, the number of applications of the procedure, that for any graph T_i, all of the components of T_i are rooted, and for any rooted component C it is the case that $w < k^\beta - 1$, $k \geq 2$ and $w \geq 1$, where k is the size of C and w is its weight. This together with the fact that eventually every component C has weight greater than W completes the proof.

As discussed before, each component C of T_0 has a root and has size k at least 2. The component C consists of at most $k-1$ edges with label 1, so its weight is at most $k-1$. Hence, the base case holds since $\beta > 1$.

Since T_0 is a subgraph of T_i which also includes all nodes of T_i, it follows that the size and weight of any rooted component of T_i are at least 2 and 1, respectively. Now, suppose the procedure adds an edge of label q from component C_1 of size k_1 and weight w_1 to component C_2 of size k_2 and weight w_2. By step 1, the new edge emanates from a minimal weight component, so $w_1 \leq w_2$. The weight w of the newly formed component is $w_1 + w_2 + q$, and the number of nodes k is $k_1 + k_2$. We show now that $w < k^\beta - 1$.

By Step 2 of the procedure, $w_1 \leq W$. Hence, it follows from Lemma 5.2 that there exist positive integers z' and q' that satisfy the zigzag inequalities and $q' = \lceil w_1(n-t)/((1-\varepsilon)t)\rceil$. Hence by Lemma 5.1 there is an edge of label q' from a root of C_1 to C_2. Thus, by the minimality of q (the weight of the edge in step 2), it follows that $q \leq q'$ which implies that $q \leq w_1(n-t)/((1-\varepsilon)t) + 1$. Let $\mathbf{r} = (n-t)/((1-\varepsilon)t) + 1$. Then,

$$w = w_1 + w_2 + q \leq \mathbf{r}\ w_1 + w_2 + 1. \tag{8}$$

Let $k_1{'}$ and $k_2{'}$ be defined by the equalities $w_1 = k_1{'}^\beta - 1$, and $w_2 = k_2{'}^\beta - 1$. Then $k_1' \leq k_2'$. We claim that

$$\begin{aligned} \mathbf{r}\ w_1 + w_2 + 1 &= \mathbf{r}\ (k_1'^\beta - 1) + (k_2'^\beta - 1) + 1 = \mathbf{r}\ k_1'^\beta + k_2'^\beta - \mathbf{r} \\ &< \mathbf{r}\ k_1'^\beta + k_2'^\beta - 1 \leq (k_1' + k_2')^\beta - 1. \end{aligned} \tag{9}$$

Using the fact that $\mathbf{r} > 1$, everything except the rightmost inequality in Equation (9) is immediate. To prove this last inequality, observe that $2^\beta = \mathbf{r} + 1$, hence equality holds for $k_1' = k_2'$. As k_2' is increased to be larger than k_1', the right side increases more rapidly than the left side since $\beta > 1$; hence, the inequality holds. Finally, by the induction hypothesis, $k_1' < k_1$, and $k_2' < k_2$. Hence,

$$(k_1' + k_2')^\beta - 1 < (k_1 + k_2)^\beta - 1 = k^\beta - 1. \tag{10}$$

Putting equations (8)$-$(10) together gives $w < k^\beta - 1$, or equivalently $k > (w+1)^\alpha$. This completes the proof of the lemma. □

Theorem 5.1 follows immediately from Lemma 5.4.

5.4. Remarks. Corollary 5.1 gives the bound we obtain in the case that $t = \Omega(n)$. However, when $t = O(n^\xi)$ for $\xi < 1$ a constant, one gets that our lower bound $(W+1)^\alpha = O(1)$ and hence is not interesting. One might wonder whether this defect results simply from the various approximations we made in proving Theorem 5.1. This seems not to be the case but is rather a limitation of our proof technique. When $t = O(n^\xi)$, the length of the sequence of graphs $\{T_0, T_1, \ldots\}$ is bounded by a constant, so the size of the largest component of the last element T is also a constant. This remains true even if one uses the least q satisfying the zig-zag inequalities rather than the q guaranteed by Lemma 5.2. Hence, to obtain a non-trivial lower bound in these cases will require either a better bound on the value q in Step 2 of the procedure than can be obtained from the zig-zag inequalities, or else a whole new proof technique. This also leaves open the possibility that Theorem 4.4 can be substantially improved. Finally, observe that the only place where we used the assumption $\tau > \lceil n/3\rceil$ is in the last paragraph of the proof of Lemma 5.2, where it is used to prove that $\tau > \lceil t/2\rceil$. Thus, our results remain correct under the weaker restriction $\tau > \lceil t/2\rceil$.

6. Relation to Other Problems. In this section we show that there are efficient reductions between the wakeup problem for $\tau = \lfloor n/2\rfloor + 1$ and the consensus and leader election problems. Hence, the wakeup problem can be viewed as a basic problem that captures the inherent difficulty of these two problems.

Lemma 6.1: *In any t-resilient consensus (leader election) protocol, a process decides (is elected) only when at least $t+1$ processes are awake.*

Proof: We first prove the lemma for a consensus protocol. Assume to the contrary that in some consensus protocol, there exists a process q and there exists a run, say ρ, in which q decides and yet no more than t processes participate in ρ. Let us assume w.l.o.g. that in ρ, process q decides on 0, and that the value of the shared register is a. Let Q be the set of processes that do not participate in ρ. Clearly, $|Q| \geq n - t$.

We can now construct a new run in which all processes in Q are correct, the initial value of the shared register is a, only processes in Q participate in it, and all processes in Q read the input value 1. Since the protocol can tolerate up to t failures this run has a prefix, say ρ', in which all processes decide. The processes that participate in ρ' must decide on the value 1 since this prefix can be extended to a run where all the n processes read the value 1 (and hence, according, to the definition of the consensus problem must decide on 1). Since the sets of processes which participate in ρ and ρ' are disjoint, and the value of the shared register at the end of ρ is the same as its value at the beginning of ρ', the composition $\langle \rho; \rho' \rangle$ is a run. However, this leads to a contradiction since processes decide on both zero and one at the same run.

The proof for a leader election protocol is similar. Assume to the contrary that in some leader election protocol, there exists a process q and there exists a run, say ρ, in which q is elected and yet no more than t processes participate in ρ. Let us assume w.l.o.g. that in ρ, the value of the shared register is a. Let Q be the set of processes that do not participate in ρ. Clearly, $|Q| \geq n - t$.

We can now construct a run in which all processes in Q are correct, in which the initial value of the shared register is a, and only processes in Q participate in it. Since the protocol can tolerate up to t failures this run has a prefix, say ρ', in which some process $q' \neq q$ is elected and writes 1 in its output register (here we used the fact that a process can be elected in a run only if it participates in this run). Clearly, the composition $\langle \rho; \rho' \rangle$ is a run. However, this leads to a contradiction since two processes are elected at the same run. □

The following theorem shows that in order to decide on some value (to be elected) in a t-resilient consensus (leader election) protocol, it is necessary to learn first that at least $t + 1$ processes are awaken. It also shows that in certain cases learning that $t + 1$ processes are awaken is sufficient for making a decision. The assumption that the processes are symmetric, is not used in the proof of that theorem.

Theorem 6.1: *(1) Any t-resilient consensus (leader election) protocol is a t-resilient wakeup protocol for any $\tau \leq t + 1$ and $p = n - t$ (p=1); (2) For any $t < n/2$, there exist t-resilient consensus and leader election protocols which are not t-resilient wakeup protocols for any $\tau \geq t + 2$.*

Proof: We prove the first part of the theorem. Assume to the contrary that for some consensus (leader election) protocol there exists a process q and there exists a run in which q decides (q is elected), and q does not know that at least $t + 1$ processes are awake. Hence, there exists a process q and there exists a run, say ρ, in which q decides (q is elected) and yet not more than t processes participate in ρ. However, this contradicts Lemma 6.1.

We now prove the second part of the theorem. We first prove the second part for a consensus protocol and then explain why it also holds for a leader election protocol. We show a consensus protocol in which, in certain n-fair run, all the processes decide on some value and yet no process ever knows (at this run) that more than t+1 processors are awaken. The protocol uses values (r, x, y, z), where $r \in \{0, ..., m-1\}$, where $m = n^2 - 2nt$, and x, y and z each belongs to $\{0, 1\}$ (note that, since $t < n/2$, we have that $m \geq n$). For integers $0 \leq i, j \leq m - 1$, let $[i, j]$ denote the cyclic interval $[i, i \oplus 1, \cdots, j]$, where $\oplus$ denotes addition modulo m. For $k = 0, \cdots, n - 1$, the cyclic intervals $[k(n - 2t), k(n - 2t) \oplus (t + 1)]$ are called *critical intervals* (Note that critical intervals may overlap).

The three bits are used as follows: x is flipped exactly once when the decision is made, y is used to hold the decision value, z is used to signal processes that wake up after the decision is made that a decision has already been reached and hence they should decide on the value in y.

Each process initially records the value of r in a local variable $init$, and then increments r by 1. Also it records the value of x, and sets $z = 0$. On alternate steps, it polls r,x and z to see if they have changed. If either x or z has changed, then a decision has already been reached and the decision value is in y, in which case the process decides on y, and thereafter sets $z = 1$ on each step. Otherwise, if it realizes that the cyclic interval $[init, r]$ includes a critical interval then it becomes the "decider" process. It then stops running the wakeup protocol, chooses its input as the consensus value, and simultaneously flips x, writes the consensus value to y, and sets $r = 1$. All of this is done with a single read-modify-write. Thereafter it sets $z = 1$ on each step.

The proof that this works is as follows. Until some process becomes the "decider", every process runs the protocol and no process changes x nor writes 1 to z. Since the longest cyclic intervals which do not contain a critical interval are of length $n - t - 1$, we have that every cyclic interval $[i, i \oplus j]$ where $n - t \leq j \leq n$ contains a critical interval. Therefore, some process eventually polls r such that the cyclic interval $[init, r]$ of this process includes a critical interval. This process becomes a decider. Every other process that waked up before the decision was made will see on its next step that x has changed and hence no such process will also become a decider. Since fewer than $n-t$ processes wake up after the decision has been made, and they are the only ones now affecting r, r is incremented by less than $n-t$ after a decision is made. Since r is set to 1 by the decider, it must be incremented by at least $n-t$ in order for any of these late processes to become a decider. Thus, none of these processes becomes a decider, and hence there is a unique decider. It follows that x and y are written to exactly once as desired, so every process that decides on something chooses the same value.

It remains to show that every process decides. Each process that wakes up before the consensus value has been chosen is either the decider or learns the consensus value on its next step thereafter, for it will see that x has changed. Since there are more than t such processes, at least one non-failing process learns the consensus value, and that process writes 1 to z infinitely many times. Since 0 is written to z at most n times, z eventually stabilizes to 1. Thereafter, every process that has not already decided sees $z=1$ and decides on its next step.

Clearly, the above protocol also solves the leader election problem, since the process called the "decider" is the elected leader, and every other process when it learns that a decision has been made knows that it can not be elected. Finally, note that when the initial value of r is $k(n-2t)$ for some k, it is possible to reach a decision when only $t+1$ processes are awake. □

Corollary 6.1: *There is no consensus or leader election protocol that can tolerate $\lceil n/2 \rceil$ failures.*

Proof: Consider a $(n-t)$-fair run in which only $n-t$ processes are awake. By the first part of Theorem 6.1, when a decision is made (a leader is elected) in this run, at least $t+1$ processes are awake. Hence $n-t \geq t+1$, which implies the corollary. □

Theorem 6.2: *Any protocol that solves the wakeup problem for any $t < n/2$, $\tau > n/2$ and $p = 1$, using a single v-valued shared register, can be transformed into a t-resilient consensus (leader election) protocol which uses a single $8v$-valued ($4v$-valued) shared register.*

Proof: First we show a reduction from the consensus problem to the wakeup problem. Suppose the wakeup solution uses values $1, ..., v$. The consensus protocol uses values (r, x, y, z), where $r \in \{1, ..., v\}$ and x, y and z each belongs to $\{0, 1\}$. The three bits are used as follows: x is flipped exactly once when the decision is made, y is used to hold the decision value, z is used to to signal processes that wake up after the decision is made that a decision has already been reached and hence they should decide on the value in y.

Each process initially stores the value of x and sets $z=0$. It then begins running the wakeup protocol. On alternate steps, it polls x and z to see if they have changed. If either has changed, then a decision has already been reached and the decision value is in y, in which case the process abandons whatever else it was doing, decides on y, and thereafter sets $z=1$ on each step. Otherwise, it continues running the wakeup protocol. If it learns that more than $n/2$ processes have waked up, if x and z still have not changed, then it becomes the "decider" process. It then stop running the wakeup protocol, chooses its input as the consensus value, and simultaneously flips x and writes the consensus value to y. All of this is done with a single read-modify-write. Thereafter it sets $z=1$ on each step.

The proof that this works is fairly straightforward and is similar to the proof of the previous theorem. Until some process decides, every process runs the wakeup protocol and no process changes x nor writes 1 to z. Hence, eventually some process will learn that more than $n/2$ processes have waked up, and that process will become a decider. Every other process that waked up before the decision was made will see on its next step that x has changed and will abandon the wakeup protocol; hence no such process will become a decider. Since fewer than $n/2$ processes wake up after the decision has been made, and they are the only ones now affecting r, none of them will learn that more than $n/2$ processes have waked up until they see $z=1$. Hence, none of them will become a decider, so there is a unique decider. It follows that x and y are written to exactly once as desired, so every process that decides on something chooses the same value.

It remains to show that every process decides. Each process that wakes up before the consensus value has been chosen is either the decider or learns the consensus value upon seeing that x has changed. Since there are more than $n/2$ of such processes, at least one non-failing process learns the consensus value, and that process writes 1 to z infinitely many times. Since 0 is written to z at most n times, z eventually stabilizes to 1. Thereafter, every process that has not already decided sees that $z = 1$ and decides on its next step.

Clearly the above reduction can be used, with minor modifications, as a reduction from the leader election problem to the wakeup problem. That is so, since the process that becomes the "decider" is the elected leader and every other process when it learns that a decision has been made knows that it can not be elected. Finally the bit y, which is used to hold the decision value, is not needed in this reduction, and hence it is sufficient to have a single $4v$-valued register. $\square$

Corollary 6.2: *(1) There is a $(\lceil n/2 \rceil - 1)$-resilient consensus (leader election) protocol that uses a single $8n$-valued ($4n$-valued) shared register, and (2) for any $t < n/2$ there is a t-resilient consensus (leader election) protocol that uses $O(t)$-valued shared register.*

Proof: From Theorem 6.2, and Theorems 4.3 and 4.4 . $\square$

The constants in Corollary 6.2 can be improved. In fact we have designed a $(\lceil n/2 \rceil - 1)$-resilient consensus (election) protocol that uses a single $3n$-valued ($2n$-valued) shared register. Next we show that the converse of Theorem 6.2 also holds. That is, the existence of a t-resilient consensus or leader election protocol which uses a single v-valued shared register, implies the existence of a t-resilient wakeup protocol for $\tau = \lfloor n/2 \rfloor + 1$, which uses a single $O(v)$-valued shared register. The idea of the proof is based on the following observation.

Lemma 6.2:

1. *Let ρ and ρ' be two runs of the same consensus protocol where at least one process decides both in ρ and in ρ'; all the processes in ρ have the same input value, say a, and when the first process decides (in ρ) it writes to the shared register some value, say c; all the processes in ρ' have the same input value, say $b \neq a$, and the run ρ' starts such that c as the value of the shared register. Let n_ρ (resp. $n_{\rho'}$) be the numbers of processes that are awake in ρ (resp. ρ') when the first process decides. Then $n_\rho + n_{\rho'} > n$.*
2. *Let ρ and ρ' be two runs of the same election protocol where some process is elected both in ρ and in ρ'; when a process is elected in ρ the shared register has some value, say c; the run ρ' starts with c as the initial value of the shared register. Let n_ρ (resp. $n_{\rho'}$) be the numbers of processes that are awake in ρ (resp. ρ') when a process is elected. Then $n_\rho + n_{\rho'} > n$.*

Proof: We start by proving the first part. Assume to the contrary that for some ρ and ρ' as above, $n_\rho + n_{\rho'} \leq n$. We can construct an n-fair run in which initially n_ρ processes behave as in ρ, until the first of them decides on a. (Note that according to the definition of the consensus problem it has to decide on a.) At this point put long delays on these processes, and let different $n_{\rho'}$ processes behave as in ρ', until someone decides on b. This leads to a contradiction since processes decide on different values at the same run.

The proof of the second part is similar. Assume to the contrary that for some ρ and ρ' as above, $n_\rho + n_{\rho'} \leq n$. We can construct a n-fair run in which initially n_ρ processes behave as in ρ, until a process is elected. At this point put long delays on these processes, and let different $n_{\rho'}$ processes behave as in ρ', until another process is elected. This leads to a contradiction since two processes are elected at the same run. $\square$

Theorem 6.3: *Any t-resilient protocol that solves the consensus or leader election problem using a single v-valued shared register can be transformed into a t-resilient protocol that solves the wakeup problem for any $\tau \leq \lfloor n/2 \rfloor + 1$ which uses a single $4v$-valued shared register.*

Proof: We show only the reduction from the wakeup problem to the consensus problem. The reduction from the wakeup problem to the leader election problem is almost the same, and the correctness proofs of the two reductions differ only by using different part of Lemma 6.2. Suppose the consensus solution uses values $1, ..., v$. The wakeup solution based on it uses values (r, x), where $r \in \{1, ..., v\}$ and $x \in \{0, 1, 2, 3\}$. Informally, this protocol works as follows. The processes run the consensus protocol such that each process considers the value of $x \pmod 2$ as its input. The first process to decide (while simulating the consensus protocol), increments x by 1 (mod 4), and each process that notices that x has been incremented by 1 restarts the simulation with the new input (i.e., $x \pmod 2$). A process that notices that x has been incremented twice or more, realizes that at least two such simulations have been completed and, by Lemma 6.2, it knows that a majority of processes are awake.

Following is a detailed description. Each process initially stores the value of the shared register (i.e., of r and x). It then begins running the consensus protocol using $x \pmod 2$ as its input. On alternate steps, in a single read-modify-write instruction, it polls r and x and behaves as follows.

- If it notices that x has been incremented twice or more since the very beginning of the simulation, then the process abandons whatever else it was doing, and terminates. As we prove later, at that point the process knows that a majority of processes are awake and hence fulfills the requirements of the wakeup problem.
- Otherwise, if x has not been changed since its previous step then the process continues running the consensus protocol (by taking one more step). Otherwise, it restart the simulation of the consensus protocol taking $x \pmod 2$ as its input value. In case that by simulating the consensus protocol the process reaches a decision (in the consensus protocol), then it increments x by 1.

We now give a correctness proof of the reduction. The notion *a round of a run* corresponds to a portion of a run between two successive changes of x. The first round is the portion from the beginning of the run until the first time x is incremented. Note that after the last round (if such a round exists) the processes may still continue running forever but, by definition, the value x is never changed thereafter. A process *participates* in a given round if during this round it simulates a step of the consensus protocol.

It should be noted that it is not clear that each round in a given run (of the simulation) corresponds to a possible execution of the consensus protocol, since some processes may participate in more than one round. In order for a round to correspond to a possible execution of the consensus protocol, it is sufficient to show that each process that participates in a round does not participate in any previous round in which $x \pmod 2$ had the same value. This clearly holds in the first two rounds. The next two claims show that this holds in the first four rounds. All the following claims refer to some specific (but arbitrary) infinite run of the reduction, in which at least $n - t$ processes are awake.

1. *Let S_i be the set of processes participating in round i. Then $S_1 \cap S_3 = \emptyset$ and $S_2 \cap S_4 = \emptyset$.* A process that participates in round i ($i \in \{1, 2\}$) will notice at round $i + 2$ (if it is given a chance to precede) that x has been incremented twice and hence will terminate without participating in round $i + 2$.
2. *Each of the first four rounds corresponds to a prefix of a run of the consensus protocol.* This claim follows from the previous one since no process participates in both rounds 1 and 3, or in both rounds 2 and 4.
3. *Once a process notices that x has been incremented twice or more, it knows that a majority of processes are awake.* By (2), we know that once x has been incremented twice or more, a simulation of at least two prefixes of runs of the consensus protocol, both satisfying the conditions of Lemma 6.2 have occurred, and hence by Lemma 6.2 a majority of processes have waken up.
4. *In every run there are at least two rounds.* A process terminates the simulation of the consensus protocol, only when it learns that x has been incremented twice or more. Hence, at least two rounds are guaranteed to be completed.
5. *There are at most three rounds in each run.*
   Assume that four rounds are completed in some run. Let n_1, n_2, n_3 and n_4 be the number of processes participating in the 1st, 2nd, 3rd and 4th rounds, respectively. By (2), all four rounds correspond to possible runs of the consensus protocol in which some process decides. This implies, by Lemma 6.2, that $n_1 + n_2 > n$ and $n_3 + n_4 > n$. Hence $n_1 + n_2 + n_3 + n_4 > 2n$. On the other hand, by (1), $n_1 + n_3 \leq n$ and $n_2 + n_4 \leq n$. But this means that $n_1 + n_2 + n_3 + n_4 \leq 2n$. A contradiction.
6. *Eventually a non-faulty process learns that a majority of processes have waken up.*
   By Lemma 6.1 and (2), in each round at least $t + 1$ processes participate, and in particular a non-faulty process participates in each round (that may be the same process). We know by (4) and (5) that in each run x is incremented either two or three times. Hence the non-faulty process that participates in the first round will eventually notice that x has been incremented two or three times, and by (3) it will know that a majority of processes are awake.

This completes the proof of the theorem. □

We notice that with one additional bit, it is possible to inform everybody that a majority of processes are awake. It follows from Theorem 6.3 that the lower bound we proved on the previous section for the wakeup problem when $\tau \geq \lfloor n/2 \rfloor + 1$ also applies to the consensus problem.

Corollary 6.3: *Let P be a t-resilient consensus or leader election protocol, and let V be the set of shared memory values used by P. Let W and α be defined as in Section 5. Then $|V| \geq (W+1)^{\alpha}/4$.*

Proof: Immediately from Theorem 5.1 and Theorem 6.3 . □

Corollary 6.4: *There is a t-resilient consensus protocol that uses $O(v)$-valued shared register iff there is a t-resilient leader election protocol that uses $O(v)$-valued shared register.*

Proof: Immediately from Theorem 6.2 and Theorem 6.3 . □

7. Conclusions. We study the new wakeup problem in a new model where 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 results are interesting for several reasons:

- They give a quantitative measure of the cost of fault-tolerance in shared memory parallel machines in terms of communication bandwidth.
- They apply to a model which more accurately reflects reality.
- They relate recent results from three different active research areas in parallel and distributed computing, namely:
  - Results in shared memory systems [Blo87, DGS88, FLBB89, Her91, Hem89, Lam86, LA87, LF83, LP90, Mis91, Tau89a, TM89, VA86].
  - The theory of knowledge in distributed systems [CM86, DM86, FHV84, FI86, FI87, FZ88, Hal86, Had87, HF89, HM90, HZ87, KT86, Leh84, Maz89, MT88, MT91, PR85, Mic89, Tut90].
  - Self-stabilizing protocols [BGW89, BP89, Dij74, Dij86, DIM90, Gou89, Kru79, KK90, Tau89b].
- They give a new point of view and enable a deeper understanding of some classical problems and results in cooperative computing.
- They are proved using techniques that will likely have application to other problems in distributed computing.

Acknowledgement. We thank Joe Halpern for helpful discussions, and the anonymous referee for very constructive comments.

REFERENCES

[Abr88] K. Abrahamson. On achieving consensus using shared memory. In *Proc. 7th ACM Symp. on Principles of Distributed Computing*, pages 291–302, 1988.

[AG85] Y. Afek and A. Gafni. Time and message bounds for election in synchronous and asynchronous complete networks. In *Proc. 4th ACM Symp. on Principles of Distributed Computing*, pages 186–195, 1985.

[BGW89] G. M. Brown, M. G. Gouda, and C.-L. Wu. Token systems that self-stabilize. *IEEE Trans. on Computers*, 38(6):845–852, June 1989.

[Blo87] B. Bloom. Constructing two-writer atomic registers. In *Proc. 6th ACM Symp. on Principles of Distributed Computing*, pages 249–259, 1987.

[BP89] J. E. Burns and J. Pachl. Uniform self-stabilizing rings. *ACM Trans. on Programming Languages and Systems*, 11(2):330–344, 1989.

[CG89] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[CM86] M. Chandy and J. Misra. How processes learn. *Journal of Distributed Computing*, 1:40–52, 1986.

[CR79] E. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configuration of processes. *Communications of the ACM*, 22(5):281–283, 1979.

[DDS87] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, 1987.

[DGS88] D. Dolev, E. Gafni, and N. Shavit. Toward a non-atomic era: l-exclusion as a test case. In *Proc. 20th ACM Symp. on Theory of Computing*, pages 78–92, 1988.

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

[Dij86] E. W. Dijkstra. A belated proof of self-stabilization. *Journal of Distributed Computing*, 1:5–6, 1986.

[DIM90] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read write atomicity. In *Proc. 9th ACM Symp. on Principles of Distributed Computing*, 1990.

[DKR82] D. Dolev, M. Klawe, and M. Rodeh. An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms*, 3:245–260, 1982.

[DLS88] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[DM86] C. Dwork and Y. Moses. Knowledge and common knowledge in a Byzantine environment i: Crash failures. In *Theoretical Aspects of Reasoning about Knowledge: Proceedings of the 1986 Conference*, pages 149–169. Morgan Kaufmann, 1986.

[FHV84] R. Fagin, Y. J. Halpern, and M. Vardi. A model theoretic analysis of knowledge. In *Proc. 25th IEEE Symp. on Foundations of Computer Science*, pages 268–278, 1984.

[FI86] M. J. Fischer and N. Immerman. Foundations of knowledge for distributed systems. In *Theoretical Aspects of Reasoning about Knowledge: Proceedings of the 1986 Conference*, pages 171–185. Morgan Kaufmann, March 1986.

[FI87] M. J. Fischer and N. Immerman. Interpreting logics of knowledge in propositional dynamic logic with converse. *Information Processing Letters*, 25(3):175–181, May 1987.

[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.

[FL87] G. Fredrickson and N. Lynch. Electing a leader in a synchronous ring. *Journal of the ACM*, 34:98–115, 1987.

[FLBB89] M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin. Distributed FIFO allocation of identical resources using small shared space. *ACM Trans. on Programming Languages and Systems*, 11(1):90–114, 1989.

[FLM86] M. J. Fischer, N. A. Lynch, and M. Merritt. Easy impossibility proofs for distributed consensus problems. *Journal of Distributed Computing*, 1:26–39, 1986.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[FZ88] M. J. Fischer and L. D. Zuck. Reasoning about uncertainty in fault-tolerant distributed systems. In M. Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 142–158. Lecture Notes in Computer Science, vol. 331, Springer-Verlag, 1988.

[GGK+84] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir. The NYU ultracomputer—designing an MIMD parallel computer. *IEEE Trans. on Computers*, pages 175–189, February 1984.

[Gou89] M. G. Gouda. The stabilizing philosopher: Asymmetry by memory and by action. *Science of Computer Programming*, 1989.

[Had87] V. Hadzilacos. A knowledge theoretic analysis of atomic commitment protocols. In *Proc. 6th ACM Symp. on Principles of Database Systems*, pages 129–134, 1987.

[Hal] Y. J. Halpern. personal communication.

[Hal86] Y. J. Halpern. Reasoning about knowledge: An overview. In *Theoretical Aspects of Reasoning about Knowledge: Proceedings of the 1986 Conference*, pages 1–17. Morgan Kaufmann, 1986.

[Hem89] D. Hemmendinger. Initializing memory shared by several processors. *International Journal of Parallel Programming*, 18:241–253, 1989.

[Her91] M. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 11(1):124–149, January 1991.

[HF89] Y. J. Halpern and R. Fagin. Modelling knowledge and action in distributed systems. *Distributed Computing*, 3:159–177, 1989.

[HM90] Y. J. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, July 1990.

[HS80] D. S. Hirschberg and J.B. Sinclair. Decentralized extrema-finding in circular configuration of processes. *Communications of the ACM*, 23:627–628, 1980.

[HZ87] Y. J. Halpern and L. D. Zuck. A little knowledge goes a long way: Simple knowledge-based derivations and correctness proofs for a family of protocols. In *Proc. 6th ACM Symp. on Principles of Distributed Computing*, pages 269–280, August 1987.

[KK90] S. Katz and Perry K.J. Self-stabilizing extensions for message-passing systems. In *Proc. 9th ACM Symp. on Principles of Distributed Computing*, 1990.

[KKM] E. Korach, S. Kutten, and S. Moran. A modular technique for the design of efficient leader finding algorithms. *ACM Trans. on Programming Languages and Systems*, 2, 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.

[Kru79] H. S. M. Kruijer. Self-stabilization (in spite of distributed control) in tree-structured systems. *Information Processing Letters*, 2:91–95, 1979.

[KT86] S. Katz and G. Taubenfeld. What processes know: Definitions and proof methods. In *Proc. 5th ACM Symp. on Principles of Distributed Computing*, pages 249–262, August 1986.

[LA87] C. M. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[Lam86] L. Lamport. The mutual exclusion problem: Statement and solutions. *Journal of the ACM*, 33:327–348, 1986.

[Leh84] D. Lehmann. Knowledge, common knowledge and related puzzles. In *Proc. 3rd ACM Symp. on Principles of Distributed Computing*, pages 62–67, 1984.

[LF83] N. A. Lynch and M. J. Fischer. A technique for decomposing algorithms which use a single shared variable. *Journal of Computer and System Sciences*, 27(3):350–377, December 1983.

[LP90] R. L. Lipton and A. Park. The processor identity problem. Manuscript, March 1990.

[Maz89] M. S. Mazer. A knowledge-theoretic account of negotiated commitment. Technical Report CSRI–237, Computer Systems Research Institute, University of Toronto, November 1989. PhD Thesis.

[Mic89] R. Michel. A categorical approach to distributed systems expressibility and knowledge. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 129–143, August 1989.

[Mis91] J. Misra. Phase synchronization. *Information Processing Letters*, 38:101–105, 1991.

[MT88] Y. Moses and M. R. Tuttle. Programming simultaneous actions using common knowledge. *Algorithmica*, 3:121–169, 1988.

[MT91] M. Merritt and G. Taubenfeld. Knowledge in shared memory systems. In *Proc. 10th ACM Symp. on Principles of Distributed Computing*, August 1991.

[Pea85] G. H. Pfister and et. al. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proceedings International Conference on Parallel Processing*, 1985.

[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.

[PKR84] J. Pachl, E. Korach, and D. Rotem. Lower bounds for distributed maximum-finding algorithms. *Journal of the ACM*, 31:905–918, 1984.

[PR85] R. Parikh and R. Ramanujam. Distributed processes and the logic of knowledge. In R. Parikh, editor, *Proceedings of the Workshop on Logic of Programs*, pages 256–268, 1985.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[RBJ88] A. G. Ranade, S. N. Bhatt, and S. L. Johnsson. The fluent abstract machine. Technical Report YALEU/DCS/TR–573, Department of Computer Science, Yale University, January 1988.

[Tau89a] G. Taubenfeld. Leader election in the presence of $n-1$ initial failures. *Information Processing Letters*, 33:25–28, 1989.

[Tau89b] G. Taubenfeld. Self-stabilizing Petri nets. Technical Report YALEU/DCS/TR–707, Department of Computer Science, Yale University, May 1989.

[Tau91] G. Taubenfeld. On the nonexistence of resilient consensus protocols. *Information Processing Letters*, 37:285–289, 1991.

[TKM89a] G. Taubenfeld, S. Katz, and S. Moran. Impossibility results in the presence of multiple faulty processes. In *9th FCT-TCS Conference, Bangalore, India*, December 1989. Lecture Notes in Computer Science, vol. 405 (eds.:C.E. Veni Madhavan), Springer Verlag 1989, pages 109-120.

[TKM89b] G. Taubenfeld, S. Katz, and S. Moran. Initial failures in distributed computations. *International Journal of Parallel Programming*, 18:255–276, 1989.

[TM89] G. Taubenfeld and S. Moran. Possibility and impossibility results in a shared memory environment. In *3rd International Workshop on Distributed Algorithms*, 1989. Lecture Notes in Computer Science, vol. 392 (eds.: J.C. Bermond and M. Raynal), Springer-Verlag 1989, pages 254–267.

[Tut90] M. Tuttle. Knowledge and distributed computation. Technical Report MIT/LCS/TR–477, Department of Computer Science, MIT, May 1990. PhD Thesis.

[VA86] P. M. B. Vitanyi and B. Awerbuch. Atomic shared register access by asynchronous hardware. In *Proc. 27th IEEE Symp. on Foundations of Computer Science*, pages 223–243, 1986. Errata, Ibid., 1987.