Knowledge in Shared Memory Systems[*]

Michael Merritt[†] Gadi Taubenfeld[†]

July 20, 1993

Abstract

We study the relation between knowledge and space. That is, we analyze how much shared memory space is needed in order to learn certain kinds of facts. Such results are useful tools for reasoning about shared memory systems. In addition we generalize a known impossibility result, and show that results about how knowledge can be gained and lost in message passing systems also hold for shared memory systems.

Key words: knowledge, fault-tolerance, shared memory, impossibility, lower bounds, agreement, consensus, renaming, wakeup.

1 Introduction

The importance of the notion of knowledge for analyzing distributed systems can be demonstrated by many examples from the literature. It has been used to design, specify, verify and prove lower bounds for protocols [2, 3, 4, 5, 8, 7, 9, 11, 12, 13, 14, 16, 18, 21, 22, 23, 24, 19, 29, 30]. With the exception of [8], this work has assumed a message passing model. Knowledge is used in [8] to investigate a particular problem, the wakeup problem, in the context of shared memory.

In this paper we point out the importance of the notion of knowledge for reasoning about shared memory systems, where processes communicate by reading and writing from a shared memory. In proving such results we do not refer to a specific problem but rather prove general results about the requirements needed to acquire certain kinds of states of knowledge. Such results are useful tools for reasoning about shared memory systems.

Our primary goal is to explore the relation between knowledge and space. That is, we analyze how much shared memory space is needed in order to learn certain kinds of facts. We prove two results which establish such a connection. These two results, described below, are for an asynchronous model which supports atomic read-modify-write operations.

The first result states that a protocol where two processes may be incorrect and where all correct processes must eventually know one of two disjoint stable predicates, has to use at least one non-binary shared register. Using this result we can show, as was first proved

[*] A preliminary version of this work appeared in the *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, pages 189–200, Montreal, Canada, August 1991.

[†] AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974.

in [17], that there is no asynchronous 2-resilient consensus protocol that uses only binary shared registers.

In the second result, we show that if initially there are two processes where neither knows some predicate and it is always the case that eventually both of them know the predicate, and furthermore there is a third process which eventually knows that they know the predicate, then it must be the case that more than a single binary shared register is used by the protocol. Using this result we can strengthen a result in [8]: a single shared binary register is insufficient for an even number of identical processes to execute a protocol in which one process eventually learns that at least three other processes are awake.

In addition, we prove that in shared memory systems which support only atomic read and atomic write operations and where at least one process may be incorrect, it is impossible to reach certain states of knowledge. This result generalizes an important impossibility result due to Fischer, Lynch and Paterson, and to Loui and Abu-Amara, which states that in asynchronous systems there cannot exist a consensus protocol that tolerates even a single undetectable crash failure [6, 17].

Finally, we show that Chandy and Misra's results [2] about how knowledge can be gained and lost in message passing systems also hold for shared memory systems.

2 Asynchronous Shared Memory Systems

In this section, we characterize asynchronous shared memory systems which support atomic read and atomic write operations or an atomic read-modify-write operation. We start with a formal description of the notion of a protocol.

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

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

- $read_p(r, v)$ – process p reads the value v from register r;
- $write_p(r, v)$ – process p writes the value v into register r;
- $rmw_p(r, v, v')$ – process p first reads a value v from r and then writes a value v', which can depend on v, into r. This event is called read-modify-write.

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

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

Let $x = (f, S)$ and $x' = (f', S')$ be runs. Run x' is a *prefix* of x (and x is an *extension* of x'), denoted $x' \leq x$, if S' is a prefix of S and $f = f'$. Let $\langle S; S' \rangle$ be the sequence obtained by concatenating the finite sequence S and the sequence S'. Then $\langle x; S' \rangle$ is an abbreviation for $(f, \langle S; S' \rangle)$. When $x \geq x'$, $(x - x')$ is the suffix of S obtained by removing S' from S.

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

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

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

Definition 1 *An asynchronous read-write protocol is a protocol with only read and write events whose runs satisfy axioms RW1 – RW3.*

Axioms for read and write

RW1 Let $\langle x; write_p(r, v) \rangle$ and y be finite runs where $x[p]y$.
Then $\langle y; write_p(r, v) \rangle$ is a run.

RW2 Let $\langle x; read_p(r, v) \rangle$ and y be finite runs where $x[p]y$.
Then $\langle y; read_p(r, u) \rangle$ is a run for some value u.

RW3 Let $\langle x; read_p(r, v) \rangle$ be a run.
Then $v = value(r, x)$.

RW1 means that if a write event which involves p can happen at a run, then the same event can happen at any run that is indistinguishable to p from it. *RW2* means that if a process is "ready to read" a value from some register, then an event on some other process cannot prevent it from reading, although it may prevent this process from reading a specific value which it could read previously. *RW3* means that it is possible to read only the last value that is written into a register.

Definition **2** *An asynchronous read-modify-write protocol is a protocol with only read-modify-write events whose runs satisfy axioms RMW1 – RMW3.*

Axioms for read-modify-write

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

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

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

RMW1 means that if a read-modify-write event which involves p can happen at a run, then the same event can happen at any run that is indistinguishable to p, provided that the register p accesses in that event has the same value in both runs. *RMW2* means that if a read-modify-write event which involves p can happen at a run, then some read-modify-write event in which p accesses the same register can happen at any run that is indistinguishable to p. *RMW3* means that it is possible to read only the last value that is written into a register.

3 The Knowledge Bivalent and Eventually Operators

In this section we define a few modal operators, and demonstrate their usefulness by generalizing a well known impossibility result.

3.1 Predicates

First we introduce the notion of a *predicate*. Formally, a predicate is simply a set of runs. It is convenient to think about a predicate, say "the value of r is 5", as the set of all runs in which the value of r is 5. Two predicates are *disjoint* if their intersection is empty. When speaking about predicates the connectives $\neg$(not), $\wedge$(and), $\vee$(or), and implication mean complement, intersection, union, and inclusion, respectively, and β *at* x means $x \in \beta$. Crucial to our exposition is the notion of a stable predicate. Informally, a stable predicate is a predicate that once true remains true thereafter.

Definition **3** *A predicate is stable if it is suffix closed. That is, if it holds in some run, it holds in all extensions of that run.*

Examples for stable predicates are: "the system is deadlocked", "consensus has been reached" and "more than five messages have been received".

3.2 Knowledge

We are now ready to define the notion of knowledge. Intuitively, a group of processes know a predicate at a given run if this predicate holds at all runs that the processes (together) cannot distinguish from the given one.

Definition **4** *For set of processes $\mathcal{G}$, predicate β and finite run x, we say that $\mathcal{G}$ knows β at x, denoted by $K_{\mathcal{G}}\beta$ at x, iff for all finite y such that $x[\mathcal{G}]y$, it is the case that β at y.*

When $\mathcal{G}$ is the singleton $\{p\}$, we write K_p instead of $K_{\mathcal{G}}$. Also, the notation $E_{\mathcal{G}}\beta$ is used as a shorthand for $\bigwedge_{p \in \mathcal{G}} K_p\beta$ (i.e., everybody in the set $\mathcal{G}$ knows β). Notice that when β is a stable predicate, then $K_{\mathcal{G}}\beta$ is **not necessarily** a stable predicate.[1] We refer the reader to [2, 10, 12] for a list of properties of the knowledge operator. Two basic facts about $K_{\mathcal{G}}$ that we use repeatedly are:

- If $K_{\mathcal{G}}\beta$ at x, then β at x;
- If $E_{\mathcal{G}}\beta$ at x, then $K_{\mathcal{G}}\beta$ at x.

3.3 Fairness

In order to discuss important properties of asynchronous protocols, we need the concept of a fair run. We say that process p is *enabled* at run x if there exists an event e_p such that $\langle x; e_p \rangle$ is a run. Notice that in asynchronous shared memory systems (i.e., protocols satisfying either the RW or the RMW axioms) an enabled process cannot become disabled as a result of an event which involves some other process.

Definition **5** *Let $\mathcal{G}$ be a set of processes. Run y is $\mathcal{G}$-fair iff for each $p \in \mathcal{G}$ and each run $x \leq y$, if p is enabled at x, then there exists a run y' such that $x \leq y' \leq y$ and such that some event in $(y' - x)$ involves p. Process p is correct in a run if the run is $\{p\}$-fair. A run is ℓ-fair iff at least ℓ processes are correct in it.*

A $\mathcal{G}$-fair run captures the intuition of an execution where all enabled processes which belong to $\mathcal{G}$ make progress. Notice that a $\mathcal{G}$-fair run may be either finite or infinite—if a $\mathcal{G}$-fair run is finite, then no process in $\mathcal{G}$ is enabled at the run, otherwise (when the run is infinite) each process in $\mathcal{G}$ that is enabled at all but finitely many prefixes appears infinitely often in the run. It easy to see that in asynchronous protocols, any finite run is a prefix of some $\mathcal{G}$-fair run, for any $\mathcal{G}$.

3.4 Operators

We next define modal operators, bivalent ($\triangle$) and eventually ($\Diamond_{\mathcal{G}}$). These new operators can also be defined using known temporal operators.

[1] For example, consider a protocol with the following five runs: (1) the *null* run, (2) e_p, (3) e_q, (4) e_pe_q, (5) e_qe_p. Let the predicate β be the set $\{e_p, e_pe_q\}$. (That is, β includes all the runs which involve the event e_p, and in which the event e_p does not appear after e_q.) Let $\mathcal{G} = \{p, q\}$. While β is stable, $K_{\mathcal{G}}\beta$ is not, since $K_{\mathcal{G}}\beta$ includes the run e_p and does not include the run e_pe_q.

DEFINITION **6** *Let β_1 and β_2 be disjoint predicates. The pair $\{\beta_1, \beta_2\}$ is bivalent at run x, denoted $\triangle\{\beta_1, \beta_2\}$ at x, iff there exist finite extensions y_1 and y_2 of x such that β_1 at y_1 and β_2 at y_2.*

Notice that the predicate $\triangle\{\beta_1, \beta_2\}$ includes exactly the runs which have finite extensions both in β_1 and in β_2.

DEFINITION **7** *For predicate β, set of processes $\mathcal{G}$ and finite run x, eventually β at x w.r.t. $\mathcal{G}$, denoted $\diamond_{\mathcal{G}}\beta$ at x, iff for every $\mathcal{G}$-fair run $y \geq x$ there is a finite prefix y', $x \leq y' \leq y$ such that β at y'.*

We use the following abbreviations: $\diamond_m\beta$ at x means $\diamond_{\mathcal{G}}\beta$ at x for every $\mathcal{G}$ where $|\mathcal{G}| \geq m$, and $\diamond$ means $\diamond_n$. (Recall that n is the total number of processes.) We say that a protocol achieves β in the presence of up to t crash failures if all its runs satisfy $\diamond_{(n-t)}\beta$. Some simple facts are listed below. All are universally quantified over all runs of an arbitrary protocol. Let $\beta_1, \beta_2, \beta_3$ be pairwise disjoint stable predicates.

1. $\triangle\{\beta_1, \beta_2\}$ *implies* $\neg(\beta_1 \vee \beta_2)$.
2. $\triangle\{\beta_1, \beta_2\}$ *implies* $\neg(\diamond\beta_1 \vee \diamond\beta_2)$.
3. $(\triangle\{\beta_1, \beta_2\} \wedge \triangle\{\beta_2, \beta_3\})$ *implies* $\triangle\{\beta_1, \beta_3\}$.
4. $(\diamond\triangle\{\beta_1, \beta_2\})$ *implies* $\triangle\{\beta_1, \beta_2\}$.
5. $\triangle\{\diamond\beta_1, \diamond\beta_2\}$ *iff* $\triangle\{\beta_1, \beta_2\}$.
6. $\diamond\beta_1$ *implies* $\neg\triangle\{\beta_1, \beta_2\}$.
7. $\diamond_{m-1}\beta_1$ *implies* $\diamond_m\beta_1$. $(1 < m \leq n)$
8. $\neg\triangle\{\beta_1, \beta_2\}$ *is a stable predicate.*

3.5 An Impossibility Theorem

We prove a theorem which generalizes an important impossibility result due to Fischer, Lynch and Paterson [6], and Loui and Abu-Amara [17] (see also [26]). For the rest of the section, let β_1 and β_2 be disjoint stable predicates. The theorem says that processes cannot reliably learn of the resolution of bivalent stable predicates in the presence of a single process fault.

Theorem 1 *In any asynchronous read-write protocol, for any run x, if $\triangle\{\beta_1, \beta_2\}$ at x, then for some set of processes $\mathcal{G}$ where $|\mathcal{G}| = n - 1$, $\neg\diamond_{\mathcal{G}}(K_{\mathcal{G}}\beta_1 \vee K_{\mathcal{G}}\beta_2)$ at x.*

In order to prove the theorem we first prove two lemmas. Without loss of generality, we consider in this proof only deterministic processes. That is, if $\langle x; e_p\rangle$ and $\langle x; e'_p\rangle$ are runs, then $e_p = e'_p$. When p is enabled at x, we denote by $\circ_p x$ the unique extension of x by a single event of p. Finally, by $\bar{\mathcal{G}}$ we denote the set of all processes not in $\mathcal{G}$.

The proof of the theorem relies on two lemmas, the first of which relates disjoint stable predicates, the eventually operator, the knowledge of a set of processes $\mathcal{G}$, and runs that are indistinguishable to $\mathcal{G}$.

Lemma 1 *Let x and y be finite runs and $\mathcal{G}$ be a set of processes. If $\Diamond_{\mathcal{G}} K_{\mathcal{G}} \beta_1$ at x, $x[\mathcal{G}]y$, and $value(r, x) = value(r, y)$ for every shared register r, then $\neg\Diamond\beta_2$ at y.*

Proof: Assume $\Diamond_{\mathcal{G}} K_{\mathcal{G}} \beta_1$ at x, $x[\mathcal{G}]y$, and $value(r, x) = value(r, y)$ for every shared register r. From $RW1$–$RW2$, x is a prefix of a $\mathcal{G}$-fair run z where $x[\bar{\mathcal{G}}]z$, that is, only processes in $\mathcal{G}$ take steps after the end of x. Since $\Diamond_{\mathcal{G}} K_{\mathcal{G}} \beta_1$ at x, there is a finite run $x \le x' \le z$ such that $x[\bar{\mathcal{G}}]x'$ and $K_{\mathcal{G}}\beta_1$ at x'. From $RW1$–$RW3$, $w = \langle y; (x' - x)\rangle$ is also a run. Since $K_{\mathcal{G}}\beta_1$ at x' and $x'[\mathcal{G}]w$, it has to be that β_1 at w. Since β_1 is stable and disjoint from β_2, it is the case that $\neg\Diamond\beta_2$ at y. ▮

We use the notation $(\Diamond K)_{n'}\beta$ at x as an abbreviation for $\Diamond_{\mathcal{G}} K_{\mathcal{G}}\beta$ at x for every set of processes $\mathcal{G}$ where $|\mathcal{G}| \ge n'$. Notice that it follows from Lemma 1 that for any finite runs x and y and set of processes $\mathcal{G}$ where $|\mathcal{G}| \ge n'$, if $(\Diamond K)_{n'}\beta_1$ at x, $x[\mathcal{G}]y$, and $value(r, x) = value(r, y)$ for every shared register r, then $\neg\Diamond\beta_2$ at y.

The second lemma proves that runs with bivalent stable predicates can always be extended, preserving bivalence and forcing enabled processes to take steps. The proof of Theorem 1 concludes by applying this lemma to construct an infinite, $\mathcal{G}$-fair run, in which the bivalence is never resolved.

Lemma 2 *For any finite run x and any process p, if $\Diamond_{\mathcal{G}}(K_{\mathcal{G}}\beta_1 \vee K_{\mathcal{G}}\beta_2)$ at x for every set of processes $\mathcal{G}$ where $|\mathcal{G}| \ge n - 1$, $\triangle\{\beta_1, \beta_2\}$ at x, and p is enabled at x, then there exists a finite $y \ge x$ such that $\triangle\{\beta_1, \beta_2\}$ at y and $\neg x[p]y$ (that is, p has taken a step in $(y - x)$).*

Proof: Assume to the contrary that for some run x and process p, $\Diamond_{\mathcal{G}}(K_{\mathcal{G}}\beta_1 \vee K_{\mathcal{G}}\beta_2)$ at x for every set of processes $\mathcal{G}$ where $|\mathcal{G}| \ge n - 1$, $\triangle\{\beta_1, \beta_2\}$ at x, p is enabled at x and there does not exist a finite $y \ge x$ such that $\neg x[p]y$ and $\triangle\{\beta_1, \beta_2\}$ at y.

Let x' be an extension of x where $\neg(\beta_1 \vee \beta_2)$ at x' and p is enabled at x'. Since β_1 and β_2 are stable, neither predicate holds yet in any prefix of x', and hence $\Diamond_{\mathcal{G}}(K_{\mathcal{G}}\beta_1 \vee K_{\mathcal{G}}\beta_2)$ at x'. Since $\neg x[p]\circ_p x'$, from the assumption we have $\neg\triangle\{\beta_1, \beta_2\}$ at $\circ_p x'$, and it follows that either $(\Diamond K)_{n-1}\beta_1$ at $\circ_p x'$ or $(\Diamond K)_{n-1}\beta_2$ at $\circ_p x'$. Assume without loss of generality that $(\Diamond K)_{n-1}\beta_1$ at $\circ_p x$.

Since $\triangle\{\beta_1, \beta_2\}$ at x and β_1 and β_2 are disjoint, there must be a first point at which the choice between β_1 and β_2 is made: there exists a finite extension z of x ($z \ne x$) such that $\Diamond\beta_2$ at z and for any $x \le y < z$ it is the case that $\triangle\{\beta_1, \beta_2\}$ at y. It is important to notice that $(\Diamond K)_{n-1}\beta_2$ at z (and hence also $(\Diamond K)_{n-1}\beta_2$ at $\circ_p z$, if $\circ_p z$ exists).

Let z' be the longest prefix of z such that $x[p]z'$. (That is, there are no steps by p between the end of x and the end of z', which implies that either $z = z'$ or $z = \circ_p z'$.) We notice that either $\Diamond\beta_2$ at z' (when $z' = z$) or $(\Diamond K)_{n-1}\beta_2$ at $\circ_p z'$, and in either case, $\Diamond\beta_2$ at $\circ_p z'$.

Consider the extensions of x which are also prefixes of z'. (See Figure 1.) Since $(\Diamond K)_{n-1}\beta_1$ at $\circ_p x$, by the observations made so far there must exist extensions y and y' (of x) where y is a one event extension of y', such that $(\Diamond K)_{n-1}\beta_1$ at $\circ_p y'$, either $\Diamond\beta_2$ at y (when $y = z$) or $(\Diamond K)_{n-1}\beta_2$ at $\circ_p y$, and in either case, $\Diamond\beta_2$ at $\circ_p y$. Let p' be the process that takes a step between y' and y. That is, $y = \langle y'; e_{p'}\rangle$ for some event $e_{p'}$ where $p' \ne p$.

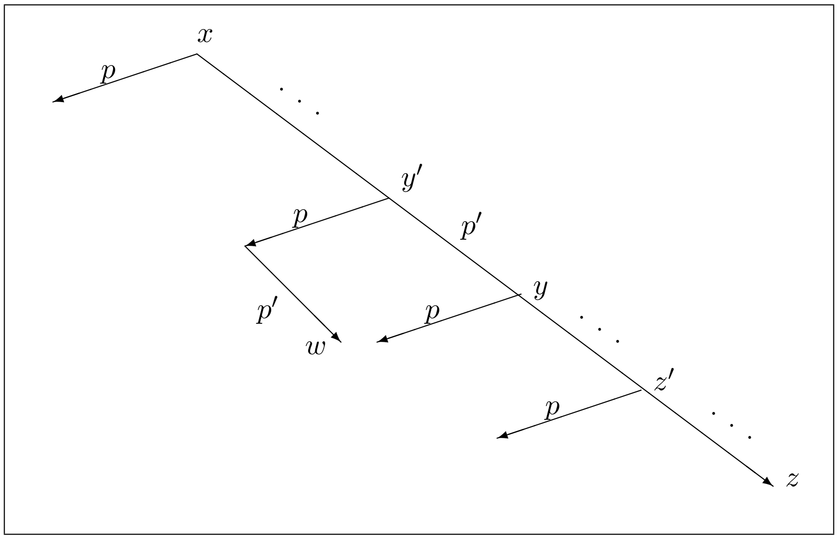

Figure 1: The runs constructed in the proof of Lemma 2.

The proof concludes by examining the event $e_{p'}$ and the possible next events by p, considering whether they are read or write events. In each case we use Lemma 1 to derive a contradiction.

First we show that $e_{p'}$ is a write event. Assume that $e_{p'}$ is not a write event. By $RW1$–$RW3$, $(\circ_p y' - y') = (\circ_p y - y)$ and hence $\circ_p y'[N - \{p'\}]\circ_p y$. Also, the values of all shared registers are the same in $\circ_p y$ and $\circ_p y'$, and as already mentioned $(\Diamond K)_{n-1}\beta_1$ at $\circ_p y'$. By Lemma 1, $\neg\Diamond\beta_2$ at $\circ_p y'$, a contradiction. Therefore, $e_{p'}$ must be a write event.

We notice that, by $RW1$ $w = \langle \circ_p y'; e_{p'}\rangle$ is a run. Also, since $(\Diamond K)_{n-1}\beta_1$ at $\circ_p y'$, it must be the case that for $\mathcal{G} = N - \{p'\}$, $\Diamond_{\mathcal{G}} K_{\mathcal{G}}\beta_1$ at w.

Next we show that $(\circ_p y' - y')$ is a write event. Assume $(\circ_p y' - y')$ is not a write event. Then, $w[N - \{p\}]y$, and $w[N - \{p\}]\circ_p y$. Also, the values of all shared registers are the same in w, y and $\circ_p y$. Because $w \geq \circ_p y'$, $\Diamond\beta_1$ at w. Now, recall that either β_2 at y or $(\Diamond K)_{n-1}\beta_2$ at $\circ_p y$. It follows that, either $(\Diamond K)_{n-1}\beta_2$ at y or $(\Diamond K)_{n-1}\beta_2$ at $\circ_p y$. By Lemma 1, in both cases $\neg\Diamond\beta_1$ at w, a contradiction. Therefore, for registers r_1 and r_2, and values v_1 and v_2, $(\circ_p y' - y') = write_p(r_1, v_1)$, and $(y - y') = write_{p'}(r_2, v_2)$.

Assume $r_1 \neq r_2$. Since the two write events are independent, the values of all shared registers are the same in w and $\circ_p y$. Also, $w[N]\circ_p y$ and hence $w[\mathcal{G}]\circ_p y$ for $\mathcal{G} = N - \{p'\}$. As pointed out, $\Diamond_{\mathcal{G}} K_{\mathcal{G}}\beta_1$ at w, and hence by Lemma 1, $\neg\Diamond\beta_2$ at $\circ_p y$, a contradiction.

Assume $r_1 = r_2$. Clearly, $value(r_1, \circ_p y') = value(r_1, \circ_p y)$. Hence, the values of all shared registers are the same in $\circ_p y'$ and $\circ_p y$. Also, $\circ_p y'[N - \{p'\}]\circ_p y$ and as assumed $(\Diamond K)_{n-1}\beta_1$ at $\circ_p y'$. As before by Lemma 1, $\neg\Diamond\beta_2$ at $\circ_p y$, a contradiction. ▮

Proof of Theorem 1: Assume to the contrary that for some run x where $\triangle\{\beta_1, \beta_2\}$ at x, it is the case that $\Diamond_{\mathcal{G}}(K_{\mathcal{G}}\beta_1 \vee K_{\mathcal{G}}\beta_2)$ at x for every set of processes $\mathcal{G}$ where $|\mathcal{G}| = n - 1$. Using Lemma 2 we inductively construct an n-fair run, $\mathcal{F}$, extending the run x and such that $\triangle\{\beta_1, \beta_2\}$ holds at all the finite prefixes of $\mathcal{F}$.

The construction of $\mathcal{F}$ is done in steps. $\mathcal{F}_0 = x$.

At each step $m \geq 1$ we extend the run $\mathcal{F}_{m-1}$ constructed at step $m-1$ to a run $\mathcal{F}_m$ as follows. If process $p_{m \pmod n}$ is not enabled at $\mathcal{F}_{m-1}$, then $\mathcal{F}_m = \mathcal{F}_{m-1}$. Otherwise, using Lemma 2, we extend $\mathcal{F}_{m-1}$ to a run $\mathcal{F}_m$ where $\neg \mathcal{F}_{m-1}[p_{m \pmod n}]\mathcal{F}_m$, $\triangle\{\beta_1, \beta_2\}$ at $\mathcal{F}_m$, and $\Diamond_{\mathcal{G}}(K_{\mathcal{G}}\beta_1 \vee K_{\mathcal{G}}\beta_2)$ at $\mathcal{F}_m$ for every set of processes $\mathcal{G}$ where $|\mathcal{G}| = n-1$.

If for some m, no process is enabled at $\mathcal{F}_m$, then $\mathcal{F}_m = \mathcal{F}$. Otherwise $\mathcal{F} = \lim_{m \to \infty} \mathcal{F}_m$.

Thus, we get that $\neg \Diamond_N(K_N\beta_1 \vee K_N\beta_2)$ at x, and hence $\neg \Diamond_{\mathcal{G}}(K_{\mathcal{G}}\beta_1 \vee K_{\mathcal{G}}\beta_2)$ at x for some set of processes $\mathcal{G}$ where $|\mathcal{G}| = n-1$, a contradiction. ▮

Recall that the the notation $E_{\mathcal{G}}\beta$ is used as a shorthand for $\bigwedge_{p \in \mathcal{G}} K_p\beta$. That is, everybody in the set $\mathcal{G}$ knows β. Moreover, $E_{\mathcal{G}}\beta$ implies $K_{\mathcal{G}}\beta$.

Corollary 1 *In any asynchronous read-write protocol, for any run x, if $\triangle\{\beta_1, \beta_2\}$ at x, then for some set of processes $\mathcal{G}$ where $|\mathcal{G}| = n-1$, $\neg \Diamond_{\mathcal{G}}(E_{\mathcal{G}}\beta_1 \vee E_{\mathcal{G}}\beta_2)$ at x.*

APPLICATION 1: In [6], it is proved that in asynchronous message passing systems there cannot exist a consensus protocol that tolerates even a single undetectable crash failure. A similar result is also proved for (shared memory) read-write protocols in [17]. As we show now, this impossibility result for shared memory systems follows easily from Corollary 1. Since asynchronous shared memory systems which support atomic read and write operations can simulate asynchronous message passing systems [17], a corresponding result follows also for asynchronous message passing systems.

A *t-resilient 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 either 0 or 1 (i.e., each correct process writes 0 or 1 into its local output register), the values written by all processes are the same, and the decision value is equal to the input value of some process.

Let β_i be the stable predicate "consensus has been reached on the value i" ($i \in \{0,1\}$). That is, β_i is the set of all runs in which at least one process writes i into its local output register. As is proved in [6] (Lemma 2) and [17] (Lemma 5.2), in any 1-resilient consensus protocol there must exist a run $x_0 = (f, \mathit{null})$ such that $\triangle\{\beta_0, \beta_1\}$ at x_0. We sketch the proof here. Let f_0 be a function which assigns to all the processes the input 0, and let f_1 be a function which assigns to all the processes the input 0. By definition, $\Diamond_{n-1}\beta_i$ at (f_i, null). Starting from f_0, by changing the input of processes from 0 to 1, one at a time, we get two functions f_0' and f_1', which differ only on the input value assigned to a single process. Since the protocol is 1-resilient it must be the case that $\triangle\{\beta_0, \beta_1\}$ at both (f_0', null) and (f_1', null). Let $x_0 = (f_0', \mathit{null})$.

Clearly, if process p decides on i at run x, then $K_p\beta_i$ at x, and thus in any t-resilient consensus protocol for any set of processes $|\mathcal{G}| \geq n-t$, $\Diamond_{\mathcal{G}}(E_{\mathcal{G}}\beta_0 \vee E_{\mathcal{G}}\beta_1)$ at x_0. Thus, by Corollary 1, the impossibility result follows.

Another application of Theorem 1 is to prove a result for the renaming problem which is defined in [1] as follows. Each of the n processes initially has a distinct name taken from an unbounded ordered domain, and the goal is to design a protocol that allows each correct process to choose irreversibly a new name from a space of size *NS*, such that every two new names are distinct. It is assumed that apart from their unique initial names the processes are identical.

It follows from Theorem 1 that there can not exist a protocol for solving the renaming problem with *NS*$= n$ which tolerates a single crash failure. The predicates β_1 and β_2 may be taken to be "process p will choose the value 1" and "process p will not choose the value 1", for some process p. (We assume that 1 belongs to the new name space.)

Similarly, it follows from Theorem 1 that there is no 1-resilient leader election protocol in such a model. A *t-resilient leader election protocol* is a protocol for n processes, where each process has local write-once output register. In any $(n-t)$-fair run of it, there exists a finite prefix in which all processes (correct or faulty) *commit* to some value in $\{0,1\}$, and exactly one process commits to 1. That process is called the *leader*. We say that a process p *commits* to a value $v \in \{0,1\}$ in x if p eventually writes v to its output register in any $(n-t)$-fair extension of x in which p is correct. (Notice that each correct process eventually knows whether it is the leader.) The predicates β_1 and β_2 may be taken to be "process p will be the leader" and "process p will not be the leader", for some process p.

The above two results about the renaming and election problems were first proved for message passing systems in [20], and for shared memory systems in [28]. (See also [1, 25, 27].) ▮

Theorem 1 states that for any asynchronous read/write protocol in which it is possible to reach either of two disjoint sets β_1 and β_2, there must be an $n-1$ fair run in which it is never possible to deduce from the local histories of the $n-1$ processes that a state belonging to a specific β_i will eventually be reached. It is natural to ask whether Theorem 1 can be strengthened to show that there is an $n-1$ fair run in which no state belonging to $\beta_1 \cup \beta_2$ is ever reached. (Is it possible to replace $\neg\Diamond_{\mathcal{G}}(K_{\mathcal{G}}\beta_1 \vee K_{\mathcal{G}}\beta_2)$ in Theorem 1, by $\neg\Diamond_{\mathcal{G}}(\beta_1 \vee \beta_2)$?)

A simple counter-example demonstrates that the resulting, stronger statement is incorrect. To get the counter-example we weaken the requirements of the leader election protocol defined above, and do not require that the elected leader or any of the other processes "knows" whether it is the elected leader. The following protocol satisfies the new specifications: There is a single shared register which each process tries to write into. The first process that succeeds to write into the register is the elected leader. Let β_1 and β_2 be the predicates: "process p is the first to write (i.e., is the leader)" and "process p is not the first to write (i.e., is not the leader)", respectively. Then clearly, initially $\triangle\{\beta_1, \beta_2\}$, and $\Diamond_{\mathcal{G}}(\beta_1 \vee \beta_2)$ at x for any run x and any (nonempty) set of processes $\mathcal{G}$.

4 Knowledge and Space

In this section we prove some lower bounds on the shared space needed to gain knowledge in asynchronous shared memory systems. Throughout this section we assume that $n \geq 3$. The following theorem states that a protocol where at least two processes may be incorrect and where all correct processes must eventually know one of two disjoint stable predicates that are possible initially, has to use at least one non-binary shared register. The theorem and its proof are inspired by Theorem 5.2 in [17]. As before, let β_1 and β_2 be disjoint stable predicates.

Theorem 2 *In any asynchronous read-modify-write protocol, for any run x where $\triangle\{\beta_1, \beta_2\}$ at x, if $\Diamond_{\mathcal{G}}(K_{\mathcal{G}}\beta_1 \vee K_{\mathcal{G}}\beta_2)$ at x for every set of processes $\mathcal{G}$ where $|\mathcal{G}| \geq n-2$, then the protocol uses at least one non-binary shared register.*

In order to prove the theorem we first prove two lemmas. In proving these lemmas we assume that all shared registers are binary registers. As in Theorem 1, without loss of generality, we consider in this proof only deterministic processes, and denote by $\circ_p x$ the unique extension of x by a single event on p. We point out that Lemma 1, and the observation following it, although proved for read-write protocols, hold also for read-modify-write protocols with essentially the same proof.

When only binary registers are available, the relative order in which two enabled processes takes read-modify-write steps can play only a limited role in what the other processes can learn. The first lemma states such a limitation.

Lemma 3 *Let x be a run, p and p' be two processes which are enabled at x, and $\mathcal{G} = N - \{p, p'\}$. If $\Diamond_{\mathcal{G}} K_{\mathcal{G}}\beta_1$ at $\circ_p x$, then $\neg\Diamond\beta_2$ at $\circ_{p'} x$.*

Proof: Assume $\Diamond_{\mathcal{G}} K_{\mathcal{G}}\beta_1$ at $\circ_p x$, and let the last step in $\circ_p x$ be $rmw_p(r_1, v'_1, v_1)$ (that is, $(\circ_p x - x) = rmw_p(r_1, v'_1, v_1)$) and similarly let the last step in $\circ_{p'} x$ be $rmw_{p'}(r_2, v'_2, v_2)$ (that is, $(\circ_{p'} x - x) = rmw_{p'}(r_2, v'_2, v_2)$), for registers r_1 and r_2 and values $v'_1, v_1, v'_2, v_2 \in \{0, 1\}$. Let $y = \circ_{p'}\circ_p x$ and $y' = \circ_p \circ_{p'} x$. Note that, since $y \geq \circ_p x$ and $y[\mathcal{G}]\circ_p x$, $\Diamond_{\mathcal{G}} K_{\mathcal{G}}\beta_1$ at y.

Assume $r_1 \neq r_2$. Since the two events are independent, the values of all shared registers are the same in y and y', and $y[N]y'$. By the note above, also $\Diamond_{\mathcal{G}} K_{\mathcal{G}}\beta_1$ at y. Thus, by Lemma 1, $\neg\Diamond\beta_2$ at y', and hence $\neg\Diamond\beta_2$ at $\circ_{p'} x$.

Assume $r_1 = r_2$. Then $v'_1 = v'_2$. Since all registers are binary, there are three possible cases. (1) $v_1 = v_2$. Since $\Diamond_{\mathcal{G}} K_{\mathcal{G}}\beta_1$ at $\circ_p x$, $\circ_p x[\mathcal{G}]\circ_{p'} x$, and $value(r, \circ_p x) = value(r, \circ_{p'} x)$ for every shared register r, by Lemma 1 $\neg\Diamond\beta_2$ at $\circ_{p'} x$. (2) $v'_1 = v_1$. By $RMW1$, $y = \langle \circ_p x; rmw_{p'}(r_2, v'_2, v_2)\rangle$ is a run. Again by the note above, $\Diamond_{\mathcal{G}} K_{\mathcal{G}}\beta_1$ at y, $y[\mathcal{G}]\circ_{p'} x$, and $value(r, y) = value(r, \circ_{p'} x)$ for every shared register r, by Lemma 1, $\neg\Diamond\beta_2$ at $\circ_{p'} x$. (3) $v'_2 = v_2$. The values of all shared registers are the same in $\circ_p x$ and y', $\circ_p x[\mathcal{G}]y'$, and also $\Diamond_{\mathcal{G}} K_{\mathcal{G}}\beta_1$ at $\circ_p x$. Thus, by Lemma 1, $\neg\Diamond\beta_2$ at y', and hence $\neg\Diamond\beta_2$ at $\circ_{p'} x$. ▮

The proof of the next lemma uses the notation $(\Diamond K)_{n'}\beta$ from the previous section. The lemma is similar to Lemma 2, showing that runs with bivalent stable predicates can always be extended, preserving bivalence and forcing enabled processes to take steps. As in the proof of Theorem 1, the proof of Theorem 2 concludes by applying this lemma to construct an infinite, $\mathcal{G}$-fair run, in which the bivalence is never resolved.

Lemma 4 *For any run x and any two processes p and p' which are enabled at x, if $\Diamond_{\mathcal{G}}(K_{\mathcal{G}}\beta_1 \vee K_{\mathcal{G}}\beta_2)$ at x for every set of processes $\mathcal{G}$ where $|\mathcal{G}| \geq n-2$ and $\triangle\{\beta_1, \beta_2\}$ at x, then there exists a finite $y \geq x$ such that $\neg x[\{p, p'\}]y$ and $\triangle\{\beta_1, \beta_2\}$ at y.*

Proof: Assume to the contrary that for some run x and two processes p and p' which are enabled at x, $\Diamond_{\mathcal{G}}(K_{\mathcal{G}}\beta_1 \vee K_{\mathcal{G}}\beta_2)$ at x for every set of processes $\mathcal{G}$ where $|\mathcal{G}| \geq n-2$, $\triangle\{\beta_1, \beta_2\}$ at x, and there does not exist $y \geq x$ such that $\neg x[\{p, p'\}]y$ and $\triangle\{\beta_1, \beta_2\}$ at y.

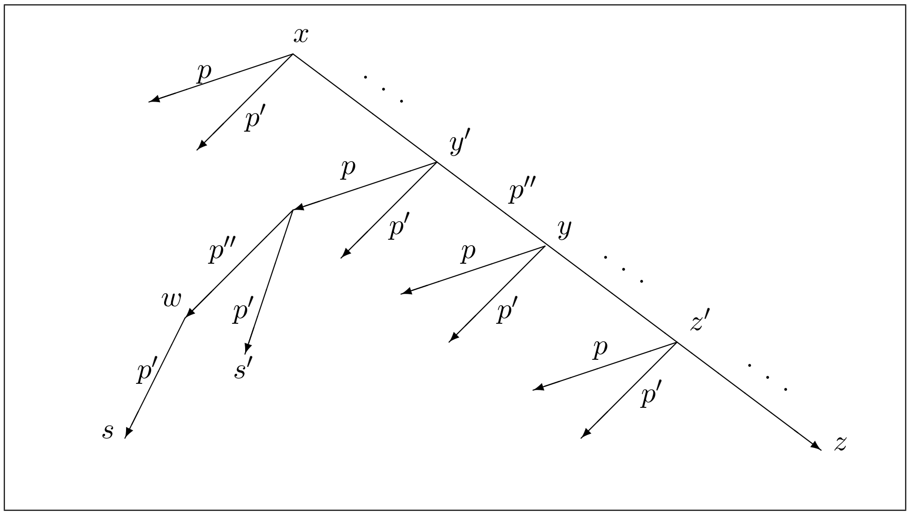

Figure 2: The runs constructed in the proof of Lemma 4.

Let x' be an extension of x where $\neg(\beta_1 \vee \beta_2)$ at x' and both p and p' are enabled at x'. Since β_1 and β_2 are stable, neither predicate holds yet in any prefix of x', and hence $\Diamond_{\mathcal{G}}(K_{\mathcal{G}}\beta_1 \vee K_{\mathcal{G}}\beta_2)$ at x'. Since $\neg x[\{p,p'\}]\circ_p x'$, from the assumption we have $\neg\triangle\{\beta_1,\beta_2\}$ at $\circ_p x'$, and it follows that either $(\Diamond K)_{n-2}\beta_1$ at $\circ_p x'$ or $(\Diamond K)_{n-2}\beta_2$ at $\circ_p x'$. Similarly, either $(\Diamond K)_{n-2}\beta_1$ at $\circ_{p'} x'$ or $(\Diamond K)_{n-2}\beta_2$ at $\circ_{p'} x'$. Since $\neg(\beta_1 \vee \beta_2)$ at x, assume without loss of generality that $(\Diamond K)_{n-2}\beta_1$ at $\circ_p x$. By Lemma 3, also $(\Diamond K)_{n-2}\beta_1$ at $\circ_{p'} x$.

Since $\triangle\{\beta_1,\beta_2\}$ at x and β_1 and β_2 are disjoint, there must be a first point at which the choice between β_1 and β_2 is made: there exists an extension z of x ($z \neq x$) such that $\Diamond\beta_2$ at z and for any $x \leq y < z$, it is the case that $\triangle\{\beta_1,\beta_2\}$ at y. It is important to notice that $(\Diamond K)_{n-2}\beta_2$ at z. Furthermore, either β_2 at z or both $(\Diamond K)_{n-2}\beta_2$ at $\circ_p z$ (if $\circ_p z$ exists) and $(\Diamond K)_{n-2}\beta_2$ at $\circ_{p'} z$ (if $\circ_{p'} z$ exists).

Let z' be the longest prefix of z such that $x[\{p,p'\}]z'$. We notice that, by RMW2, for any y such that $x \leq y \leq z'$, both p and p' are enabled at y. (See Figure 2.) Since $z' \leq z$, it follows from the first assumption and Lemma 3 that either β_2 at z' (when $z' = z$) or $(\Diamond K)_{n-2}\beta_2$ at $\circ_p z'$ and $(\Diamond K)_{n-2}\beta_2$ at $\circ_{p'} z'$. In any case, $\Diamond\beta_2$ at both $\circ_p z'$ and $\circ_{p'} z'$.

Consider the extensions of x which are also prefixes of z'. Since $(\Diamond K)_{n-2}\beta_1$ at $\circ_p x$ there must exist extensions y and y' (of x) where y is a one event extension of y', such that $(\Diamond K)_{n-2}\beta_1$ at both $\circ_p y'$ and $\circ_{p'} y'$, and either β_2 at y (when $y = z$) or $(\Diamond K)_{n-2}\beta_2$ at both $\circ_p y$ and $\circ_{p'} y$, and in either case, $\Diamond\beta_2$ at both $\circ_p y$ and $\circ_{p'} y$. Let p'' be the process that takes a step between y' and y. That is, $y = \langle y'; e_{p''}\rangle$ for some event $e_{p''}$ where $p'' \notin \{p,p'\}$.

For some registers r_1 and r_2, and values $v_1', v_1, v_2', v_2 \in \{0,1\}$, $(\circ_p y' - y') = rmw_p(r_1, v_1', v_1)$, and $(y - y') = rmw_{p''}(r_2, v_2', v_2)$.

The proof concludes by examining the events $rmw_p(r_1, v_1', v_1)$ and $rmw_{p''}(r_2, v_2', v_2)$, depending on whether $r_1 = r_2$ and the values read and written. In each case we use Lemma 1 to derive a contradiction.

Assume $r_1 \neq r_2$. By $RMW1$, $w = \langle y'; rmw_p(r_1, v_1', v_1); rmw_{p''}(r_2, v_2', v_2)\rangle$ is a run. The values of all shared registers are the same in w and $\circ_p y$, and $w[N]\circ_p y$. Let $\mathcal{G} = \{N - p''\}$, because $w > \circ_p y'$ and $w[\mathcal{G}]\circ_p y'$, $\Diamond_{\mathcal{G}} K_{\mathcal{G}} \beta_1$ at w. By Lemma 1, $\neg\Diamond\beta_2$ at $\circ_p y$, a contradiction.

Assume $r_1 = r_2$. Then $v_1' = v_2'$. Since all registers are binary, there are three possible cases.

(1) $v_2' = v_2$. Since $(\Diamond K)_{n\text{–}2}\beta_1$ at $\circ_p y'$, $\circ_p y'[N - \{p''\}]\circ_p y$, and the values of all shared registers are the same in $\circ_p y'$ and $\circ_p y$, by Lemma 1, $\neg\Diamond\beta_2$ at $\circ_p y$, a contradiction.

(2) $v_1' = v_1$. Let $\mathcal{H} = N - \{p, p'\}$. Recall that either β_2 at y or $(\Diamond K)_{n\text{–}2}\beta_2$ at $\circ_{p'} y$. It follows that, either $(\Diamond K)_{n\text{–}2}\beta_2$ at y or $(\Diamond K)_{n\text{–}2}\beta_2$ at $\circ_{p'} y$, and in either case $\Diamond_{\mathcal{H}} K_{\mathcal{H}} \beta_2$ at $\circ_{p'} y$. Let $s = \circ_{p'} \circ_{p''} \circ_p y'$. Because $s > \circ_p y'$, $\Diamond\beta_1$ at s. Since $\circ_{p'} y[\mathcal{H}]s$, and the values of all shared registers are the same in $\circ_{p'} y$ and s, by Lemma 1, $\neg\Diamond\beta_1$ at s, a contradiction.

(3) $v_1 = v_2$. Recall that either β_2 at y or $(\Diamond K)_{n\text{–}2}\beta_2$ at $\circ_{p'} y$. It follows that, either $(\Diamond K)_{n\text{–}2}\beta_2$ at y or $(\Diamond K)_{n\text{–}2}\beta_2$ at $\circ_{p'} y$. Assume $(\Diamond K)_{n\text{–}2}\beta_2$ at y. Since $y[N - \{p, p''\}]\circ_p y'$, and the values of all shared registers are the same in y and $\circ_p y'$, by Lemma 1, $\neg\Diamond\beta_1$ at $\circ_p y'$, a contradiction. Assume $(\Diamond K)_{n\text{–}2}\beta_2$ at $\circ_{p'} y$. Let $s' = \circ_{p'} \circ_p y'$. Because $s' > \circ_p y'$, $\Diamond\beta_1$ at s'. Since $\circ_{p'} y[N - \{p, p''\}]s'$, and the values of all shared registers are the same in $\circ_{p'} y$ and s', by Lemma 1, $\neg\Diamond\beta_1$ at s', a contradiction. ∎

Proof of Theorem 2: Assume to the contrary that for some run x where $\triangle\{\beta_1, \beta_2\}$ *at* x, it is the case that $\Diamond_{\mathcal{G}}(K_{\mathcal{G}}\beta_1 \vee K_{\mathcal{G}}\beta_2)$ at x for every set of processes $\mathcal{G}$ where $|\mathcal{G}| = n - 2$. We notice that since $\triangle\{\beta_1, \beta_2\}$ at x, there are at least two enabled processes in x. Using Lemma 4 we can inductively construct an $(n-1)$-fair run extending x, such that $\triangle\{\beta_1, \beta_2\}$ holds at all the finite prefixes of that run. (The inductive construction is similar to that in Theorem 1.) Thus, $\neg\Diamond_{\mathcal{G}}(K_{\mathcal{G}}\beta_1 \vee K_{\mathcal{G}}\beta_2)$ at x for some set of processes $\mathcal{G}$ where $|\mathcal{G}| = n - 2$, a contradiction. ∎

Corollary 2 *In any asynchronous read-modify-write protocol, for any run x where $\triangle\{\beta_1, \beta_2\}$ at x, if $\Diamond_{\mathcal{G}}(E_{\mathcal{G}}\beta_1 \vee E_{\mathcal{G}}\beta_2)$ at x for every set of processes $\mathcal{G}$ where $|\mathcal{G}| \geq n-2$, then the protocol uses at least one non-binary shared register.*

Application 2: Recall the definition of a t-resilient consensus protocol from the first Application section, and as before, let β_i be the stable predicate "consensus has been reached on the value i". That section also sketched a proof that there must exist a run $x = (f, null)$ such that $\triangle\{\beta_0, \beta_1\}$ at x (see also [6], Lemma 2). Clearly, if process p decides on i at run x, then $K_p\beta_i$ at x, and thus in any 2-resilient consensus protocol $\Diamond_{\mathcal{G}}(E_{\mathcal{G}}\beta_0 \vee E_{\mathcal{G}}\beta_1)$ at x for every set of processes $|\mathcal{G}| \geq n - 2$.

Since every 2-resilient consensus protocol satisfies all the premises of Corollary 2, we can conclude that: *there is no asynchronous 2-resilient consensus read-modify-write protocol that uses only binary shared registers.* This result was first proved by Loui and Abu-Amara [17] (Theorem 5.2). ∎

Corollary 2 can also be used to argue that any 2-resilient leader election protocol using read-modify-write must use at least one non-binary shared register. However, in this case we must require that every correct process will eventually know the leader's identity. Without this requirement the problem can be solved using one bit. Each process tries to write this bit, and the first to succeed is the leader. Since the operation on the bit is read-modify-write, each process learns whether or not it is the first to write, and hence knows whether or not it is the leader. But those who are not leaders do not know the leader's identity.

This example serves also as a demonstration that, as in Theorem 1, Theorem 2 cannot be strengthened by replacing $\neg\Diamond_{\mathcal{G}}(K_{\mathcal{G}}\beta_1 \vee K_{\mathcal{G}}\beta_2)$ with $\neg\Diamond_{\mathcal{G}}(\beta_1 \vee \beta_2)$. Moreover, an extension of this example illustrates why the theorem refers to runs in which as many as two processes may be faulty—if only one process might be faulty, then either the leader can reveal its identity, or the leader fails, and the other processes can identify the leader by all revealing that they are not the leader. (This can be done with binary registers.) If two processes fail, one of which is the leader, the other processes can be left with unresolvable ambiguity as to which of the failed processes is the leader.

In the next theorem, we show that if initially there are two processes where each does not know some predicate and it is always the case that eventually each one of the two does know the predicate and furthermore there is a third process which eventually knows that they do know the predicate, then it must be the case that more than a single binary shared register is used in the protocol. We assume in the following theorem a model of computation which consists of $n \geq 3$ *identical* processes. We use the notation $S^m_{\mathcal{G}}\beta$ to mean that there exist at least m distinct processes which belong to $\mathcal{G}$ that each know β. Recall that N is the set of all the processes.

Theorem 3 *In any asynchronous read-modify-write protocol,*

1. *for an even number of identical processes, if there exists a run $x = (f, null)$ and a predicate β such that $(\neg S^1_N\beta \wedge \Diamond \bigvee_{p\in N} K_p S^2_{N-\{p\}}\beta)$ at x, then either the protocol uses a non-binary shared register or it uses more than one binary shared register.*

2. *for any number of identical processes, if there exists a run $x = (f, null)$ and a predicate β such that $(\neg S^1_N\beta \wedge \Diamond \bigvee_{p\in N} K_p S^3_{N-\{p\}}\beta)$ at x, then either the protocol uses a non-binary shared register or it uses more than one binary shared register.*

Proof: We prove only the first part of the theorem. The proof of the second part of the theorem has the same flavor as that of the first part. We assume to the contrary that there exists a run $x = (f, null)$ and a predicate β such that $(\neg S^1_N\beta \wedge \Diamond \bigvee_{p\in N} K_p S^2_{N-\{p\}}\beta)$ at x and the protocol uses a single binary shared register r, and without loss of generality, assume that $f(r) = 0$.

To prove the theorem we construct an n-fair extension of x which is used to demonstrate a contradiction. For simplicity, we assume that a process can always take a step whenever it is scheduled. We say that a process p writes the value a_∞ ($a \in \{0, 1\}$) in a run x if p writes a from x (the last step in $\circ_p x$ writes a), and in the infinite extension in which this process is the only one that is activated, a appears infinitely many times. Since in any infinite run either 0 or 1 (or both) appears infinitely many times, any process that is activated alone will eventually write either 0_∞ or 1_∞ (because with read-modify-write, the process must write a value every time it takes a step).

Pick an arbitrary process (recall they are identical and deterministic), and consider the following scenario. Starting with the value b ($b \in \{0, 1\}$) in the single binary register, we start the process running alone until it first writes a_∞, for $a \neq b$. (This may never happen.) At that point we interfere and flip the shared bit so that its value is again b. Afterwards we let the process continue until it writes a_∞ again and then we flip the bit and so on. Let $flip(b\ to\ a)$ be the number of times the process writes a_∞ in such an infinite run.

We consider the following four cases:

1. $flip(0\ to\ 1) = 0$. We construct an n-fair extension z of x 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, we let it run until it writes 0_∞ (which it must, since it does not write 1_∞. Each process cannot distinguish z from the run in which it is the only process that is activated.

2. Both $flip(0\ to\ 1)$ and $flip(1\ to\ 0)$ are infinite. Starting with 0 as the initial value we alternately activate the n processes in a round robin fashion infinitely many times. Each time a process is scheduled, if the value of the shared bit is 0 (1) we let it run until it writes 1_∞ (0_∞). No process can distinguish z from the run, constructed similarly, in which only two processes participate. (This and similar arguments below depend on there being an even number of identical processes.)

3. $flip(0\ to\ 1) \leq flip(1\ to\ 0)$, and $flip(0\ to\ 1) = k$ for some positive number k. We first alternately activate the processes in a round robin fashion for k rounds. Each time a process is scheduled, if the value of the shared bit is 0 (1) we let it run until it writes 1_∞ (0_∞). After k rounds the value of the shared bit is 0. Furthermore, if activated alone from this point, each of the n processes will eventually write 0_∞. We extend this run to an n-fair run by continuing to activate the processes in a round robin fashion, letting each process make one or more steps whenever it is scheduled until it writes 0_∞. As in the previous case, no process can distinguish this run from the run, constructed similarly, in which only two processes participate.

4. $flip(1\ to\ 0) < flip(0\ to\ 1)$, and $flip(1\ to\ 0) = k$ for some positive number k. As in the previous case, we first alternately activate the processes in a round robin fashion for k rounds. Each time a process is scheduled, if the value of the shared bit is 0 (1) we let it run until it writes 1_∞ (0_∞). After k rounds the value of the shared bit is 0. Then we schedule the first process that was activated and let it run until it writes 1_∞ . Furthermore, if activated alone from this point, each of the n processes will eventually write 1_∞. We extend this run to an n-fair run by continuing to activate the processes in a round robin fashion letting each process make one or more steps whenever it is scheduled until it writes 1_∞. As in the previous case, no process can distinguish this run from the run, constructed similarly, in which only two processes participate.

In each of the four cases above (which are exhaustive), we constructed an n-fair extension of x, called z, such that each process say p cannot distinguish z from the run in which only p and at most one other process, p', is activated. Since in this latter run only p and p' participate, p can not know in z something that it did not know at x about the knowledge of processes other than p or p'. That is, for any process p and predicate β, if $\neg S^1_N \beta$ at x, then for any prefix y of z it is not the case that $K_p S^2_{N-\{p\}} \beta$ at y, and thus $\neg \Diamond \bigvee_{p \in N} K_p S^2_{N-\{p\}} \beta$ at x, a contradiction. ∎

The assumption that the number of processes n is even can be replaced with an assumption that the shared register is initially in an arbitrary unknown state, obtaining an impossibility result valid for any n. This result is a generalization of Theorem 3.3 of [8]. One of these assumptions is necessary: if there are an odd number of processes, and no uncertainty in the initial value in the register, then the first part of the theorem is not true.

Application 3: The (fault-free) wakeup problem is a deceptively simple problem. The goal is to design a protocol for n asynchronous identical processes in a shared memory environment such that at least one process eventually learns that at least τ processes have awakened and begun participating in the protocol [8].

We say that a process p is *awake* in a run if the run contains an event that involves p. The predicate "at least τ processes are awake in run x" is the set of all runs for which there exist τ different processes which are awake in the run. Let β be the predicate "at least one process is awake" and let x be some run in which no process has taken a step (i.e., $x = (f, \mathit{null})$ for some f). Then, for any wakeup protocol where $\tau \geq i$, eventually some process learns that $i-1$ other processes have awakened. That is, $(\neg S_N^1 \beta \wedge \Diamond \bigvee_{p \in N} K_p S_{N-\{p\}}^{i-1} \beta)$ at x.

Hence, since any wakeup protocol satisfies all the premises of Theorem 3, we can conclude that: *There does not exist an asynchronous read-modify-write protocol which uses only a single binary shared register that solves the wakeup problem for: (1) an even number of processes when $\tau \geq 3$; (2) any number of processes when $\tau \geq 4$.* For even n, this lower bound is stronger than that in [8] (Theorem 3.3), since it does not assume that the shared register is initially in an arbitrary unknown state. ▮

5 How Knowledge is Gained and Lost

In this section we show that some results of Chandy and Misra for message passing systems ([2]) hold also for shared memory systems. All the results in this section are for both read-write protocols and read-modify-write protocols. We first define the notion of *causality* between events along the lines suggested by Lamport for message passing systems [15]. The causality relation captures the intuition that one event may have caused by some other event. We say that e' is a *corresponding* event of e in a given run, if in e' a process reads the value of register r and e was the most recent write to r preceding e'. Notice that both e and e' may be read-modify-write events, and that several reads may correspond to a single write.

Definition **8** *Event e may have caused event e' in run x, denoted $e \xrightarrow{x} e'$, if both events appear in x and either,*
1. e' is a corresponding event of e, or
2. $e = e'$ or both events involve the same process and e appears before e' in x, or
3. there exists an event e'' such that $e \xrightarrow{x} e''$ and $e'' \xrightarrow{x} e'$.

We use the event chain $e_{p_1} \xrightarrow{x} \ldots \xrightarrow{x} e_{p_m}$ as an abbreviation for $e_{p_i} \xrightarrow{x} e_{p_{i+1}}$ for $1 \leq i < m$. (Recall from the definition that e_p is an event which involves process p.)

Definition **9** *A run x has a process chain $\langle p_1, \ldots, p_m \rangle$ if there exist events $e_{p_1}, \ldots, e_{p_m}$ in x such that $e_{p_1} \xrightarrow{x} \ldots \xrightarrow{x} e_{p_m}$.*

For $x = (f, S)$ and $y = (f', S')$, the notation $x \propto y$ means that $f = f'$, S is a subsequence of S', and for every e' in S, $e \xrightarrow{y} e'$ implies that $e \xrightarrow{x} e'$.

Lemma 5 *If y is a run, then every $x \propto y$ is a run.*

Proof: Assume $x = (f, S)$ is finite, $y = (f, S')$, and $x \propto y$. The proof is by induction on $|S|$, the length of S. When $|S| = 0$, x is a prefix of y, and hence x is a run. We assume that the theorem holds for $|S| = m - 1$ and prove it for $|S| = m$.

Let e_p be the last event in the sequence S. There must exist $x' = (f, T)$ and $y' = (f, T')$, such that $\langle x'; e_p \rangle = x$ and $\langle y'; e_p \rangle \leq y$. It follows from the fact that $x \propto y$, and the way x' and y' are constructed that $x' \propto y'$. Since $\langle y'; e_p \rangle \leq y$, both y' and $\langle y'; e_p \rangle$ are runs. Thus, by the induction hypothesis, x' is a run.

Next we show that $x'[p]y'$. Since $x \propto y$, $S_p \leq S'_p$ and hence also $T_p \leq T'_p$. (Recall that S_p is the subsequence of S containing all events in S which involve p.) For every event e'_p in y', by Definition 8, $e'_p \xrightarrow{y} e_p$, and (since $x \propto y$) e'_p appears also in x', which implies that $T'_p \leq T_p$. Thus $S_p = S'_p$, which means that $x'[p]y'$.

If e_p is a write event, then by *RW1* $\langle x'; e_p \rangle = x$ is a run.

If e_p is a read event from register r, then it follows from *RW2* that $\langle x'; \mathit{read}_p(r, \mathit{value}(r, x')) \rangle$ is a run.

If no process writes to r in y', then $\mathit{value}(r, x') = \mathit{value}(r, y')$ and hence by *RW3* $\langle x'; \mathit{read}_p(r, \mathit{value}(r, x')) \rangle = \langle x'; e_p \rangle = x$.

Assume e is the last event that affects register r in y'. Hence $e \xrightarrow{y} e_p$, and hence (since $x \propto y$) e is also the last event that affects register r in x'. This implies that $\mathit{value}(r, x') = \mathit{value}(r, y')$ and hence again by *RW3* $\langle x'; \mathit{read}_p(r, \mathit{value}(r, x')) \rangle = \langle x'; e_p \rangle = x$.

Similarly, we show that x is run when e_p is a read-modify-write event using *RMW1*. Finally, since all finite prefixes of x are runs, x is a run also when x is infinite. ▮

In the following the notion of the indistinguishability relation $[p]$ is generalized.

Definition **10** *For any runs x and z and processes $p_1, ..., p_m$, $x[p_1, ..., p_m]z$ iff there exists a run y such that $x[p_1, ..., p_{m-1}]y$ and $y[p_m]z$.*

The definition is perhaps best understood by considering a labeled undirected graph, where the nodes are the runs and there is an edge between x and z labeled by p iff $x[p]z$ holds. In that context $x[p_1, ..., p_m]z$ means that there is a path from x to z by a sequence of edges labeled $p_1, ..., p_m$. The following theorem relates the notions of process chains and indistinguishability.

Theorem 4 *For any runs x and z where $x \leq z$ and processes $p_1, ..., p_m$, either $x[p_1, ..., p_m]z$ or there is a process chain $\langle p_1, ..., p_m \rangle$ in $(z - x)$.*

Proof: The proof is adapted with minor changes from [2]. The theorem clearly holds when $x = z$, so assume x is finite. The proof is by induction on m. For $m = 1$ the absence of process chain $\langle p_1 \rangle$ implies that there is no event which involves p_1 in $(z - x)$ and hence $x[p_1]z$. We assume that the theorem holds for $m - 1$ processes and prove it for m.

Let E be the subsequence of events in $(z - x)$ consisting of all events that may be the cause of some event which involves p_m. In particular, since the "may have caused" relation

(Definition 8) is reflexive, and for every event e' in z which involves p_m, $e' \xrightarrow{y} e'$, it is the case E contains all events in $(z - x)$ which involve p_m. (E is empty iff $y[p_m]z$.)

Let $y = \langle x; E \rangle$. Since $y \propto z$, by Lemma 5 y is a run. Since the subsequence of all events which involve p_m in y and z are the same, it is the case that $y[p_m]z$.

If $x[p_1, ..., p_{m-1}]y$, then since $y[p_m]z$, by definition $x[p_1, ..., p_m]z$ and we are done. Otherwise, by the induction hypothesis, there is a process chain $\langle p_1, ..., p_{m-1} \rangle$ in $(y - x)$. Let e be the last event in the event chain implied by the existence of such a process chain. According to the construction $e \xrightarrow{z} e'$ for some event e' which involves p_m. Hence there is a process chain $\langle p_1, ..., p_m \rangle$ in $(z - x)$. ▮

The next theorem is about how knowledge can be gained and lost in (either message passing or) shared memory asynchronous systems. For process p_i, the notation $K_i\beta$ is used for $K_{p_i}\beta$. We first prove a technical lemma.

Lemma 6 *Let x and z be runs and let $p_1, ..., p_m$ be processes. If $K_1 \ldots K_m\beta$ at x and $x[p_1, ..., p_m]z$, then $K_m\beta$ at z.*

Proof: The proof is adapted with minor changes from [2]. The proof is by induction on m. For $m = 1$, $K_1\beta$ at x and $x[p_1]z$ immediately implies that $K_1\beta$ at z.

We assume that the lemma holds for $m - 1$ processes and prove it for m. From Definition 10 it follows that there is a run y such that $x[p_1, ..., p_{m-1}]y$ and $y[p_m]z$.

From $x[p_1, ..., p_{m-1}]y$ and $K_1 \ldots K_{m-1}(K_m\beta)$ at x, using the induction hypothesis, it follows that $K_{m-1}K_m\beta$ at y, and hence also $K_m\beta$ at y. Finally, since $K_m\beta$ at y and $y[p_m]z$, it follows that $K_m\beta$ at z. ▮

Theorem 5 *Let x and y be runs where $x \leq y$, and let $p_1, ..., p_m$ be processes.*
1. If $\neg K_m\beta$ at x and $K_1 \ldots K_m\beta$ at y, then there is a process chain $\langle p_m, ..., p_1 \rangle$ in $(y - x)$.
2. If $K_1 \ldots K_m\beta$ at x and $\neg K_m\beta$ at y, then there is a process chain $\langle p_1, ..., p_m \rangle$ in $(y - x)$.

Proof: 1. Since $K_1 \ldots K_m\beta$ at y and $\neg K_m\beta$ at x by Lemma 6 it must be the case that not $x[p_m, ..., p_1]y$. Thus, by Theorem 4 there is a process chain $\langle p_m, ..., p_1 \rangle$ in $(y - x)$.

2. Since $K_1 \ldots K_m\beta$ at x and $\neg K_m\beta$ at y by Lemma 6 it must be the case that not $x[p_1, ..., p_m]y$. Thus, by Theorem 4 there is a process chain $\langle p_1, ..., p_m \rangle$ in $(y - x)$. ▮

As in [2], the two theorems still hold when speaking about sets of processes rather than single processes.

6 Discussion

We have tried to point out the importance of the notion of knowledge for reasoning about shared memory systems. We consider our results a first step in better understanding the relation between knowledge and space in shared memory systems.

We introduced two results which explicitly relate knowledge and space and showed how to derive from them known and new results about consensus, leader election, and wakeup problems. In addition we generalized known impossibility results, and showed that results about how knowledge can be gained and lost in message passing systems also hold for shared memory systems.

There is still much work to be done in exploring how much shared space is required to reach different kinds of states of knowledge. Such results will hopefully unify and generalize known results for specific problems and be useful tools for reasoning about shared memory systems.

Acknowledgements We thank Yoram Moses and Lenore Zuck for pointing out an error in an earlier version of the paper, and two referees for their very constructive comments.

References

[1] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524–548, July 1990.

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

[3] 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.

[4] R. Fagin, J. Y. Halpern, and M. Vardi. A model theoretic analysis of knowledge. *Journal of the ACM*, 38(2):382–428, April 1991.

[5] 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.

[6] 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.

[7] 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.

[8] M.J. Fischer, S. Moran, S. Rudich, and G. Taubenfeld. The wakeup problem. In *Proc. 22st ACM Symp. on Theory of Computing*, pages 106–116, May 1990. See also Technion Report #644, Computer Science Department, The Technion, August 1990.

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

[10] J. Y. Halpern. Using reasoning about knowledge to analyze distributed systems. *Annual Review of Computer Science*, 2:37–68, 1987.

[11] J. Y. Halpern and R. Fagin. A formal model of knowledge, action, and communication in distributed systems: Preliminary report. In *Proc. 4th ACM Symp. on Principles of Distributed Computing*, pages 224–236, 1985.

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

[13] J. Y. Halpern and L. D. Zuck. A little knowledge goes a long way: Simple knowledge-based derivations and correctness proofs for a family of protocols. *Journal of the ACM*, 39(3):449–478, July 1992.

[14] 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.

[15] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

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

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

[18] 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.

[19] 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.

[20] S. Moran and Y. Wolfsthal. An extended impossibility result for asynchronous complete networks. *Information Processing Letters*, 26:141–151, 1987.

[21] Y. Moses and G. Roth. On reliable message diffusion. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 119–127, August 1989.

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

[23] G. Neiger and S. Toueg. Substituting for real time and common knowledge in asynchronous distributed system. In *Proc. 6th ACM Symp. on Principles of Distributed Computing*, pages 281–293, 1987.

[24] 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.

[25] G. Taubenfeld. Impossibility results for decision protocols. Technical Report 445, Technion, January 1987. Revised version appeared as Technion TR-#506, April 1988.

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

[27] 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. To appear in *Information and Computation*.

[28] 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.

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

[30] D.-W. Wang and L. D. Zuck. Tight bounds for the sequence transmission problem. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 73–83, August 1989.