INFORMATION AND COMPUTATION 119, 283–293 (1995)

Simplifying the Design of Knowledge-Based Algorithms Using Knowledge Consistency*

GIL NEIGER

Software Technology Laboratory, Intel Corporation, JF3-206, 2111 N.E. 25th Avenue, Hillsboro, Oregon 97124-5961

Processor knowledge is an important tool in the study of distributed computer systems. It has led to better understanding of existing algorithms for such systems and to the development of new *knowledge-based* algorithms. Some of these algorithms use forms of knowledge (e.g., common knowledge) that cannot be achieved in certain systems. This paper considers *alternative interpretations* of knowledge under which these forms of knowledge can be achieved. It explores *consistent knowledge interpretations* and shows how they can be used to circumvent the known impossibility results in a number of cases. This may lead to greater applicability of knowledge-based algorithms. © 1995 Academic Press, Inc.

1. INTRODUCTION

The study of knowledge in distributed computer systems has helped both in the understanding of existing systems and in the development of new distributed algorithms. This paper seeks to expand the applicability of knowledge in distributed systems by examining notions of *knowledge consistency*.

Knowledge in distributed systems was first explored by Halpern and Moses [10]. They formalized the notion of ascribing knowledge to individual processors in a system and defined a hierarchy of states of knowledge that a set of processors may possess. The highest of these is *common knowledge*. Intuitively, a fact is common knowledge to a group if everyone in the group knows it, and everyone in the group knows that everyone in the group knows it, and so on. Halpern and Moses showed that, in many systems of practical interest, common knowledge of interesting facts cannot be achieved. Following this, they defined several modifications (weakenings) of common knowledge that *can* be achieved in practical systems. They argued that, in certain cases, these weakenings of common knowledge might adequately substitute for true common knowledge.

Halpern and Fagin explored the relationship between knowledge and action in distributed systems and introduced *knowledge-based protocols* (or algorithms), which use knowledge to determine what actions processors perform [8]. The ability to specify a processor's actions with a knowledge-based algorithm can simplify the development of solutions to many problems in distributed systems. For example, Halpern and Zuck considered the *sequence transmission* problem and developed a knowledge-based solution to it that would be correct in different types of distributed systems [12]. In order to implement a knowledge-based algorithm using a normal one, one must be able to *interpret* a processor's knowledge. This can be done in a variety of different ways.

Several researchers have examined the relationship between knowledge and the solutions to specific problems in distributed systems. Dwork and Moses showed that it is necessary to attain common knowledge in order to achieve *Simultaneous Byzantine Agreement* [6]; Moses and Tuttle did the same for general simultaneous actions [18]. Neiger and Tuttle showed similar results for a related class of problems [21]. Each of these arguments included a demonstration of a knowledge-based algorithm to solve the problem being considered or a proof that no solution was possible because the requisite knowledge could not be achieved. More recently, Halpern *et al.* have explored the use of knowledge in deriving round-optimal solutions for *Eventual Byzantine Agreement* [11]. Neiger and Bazzi showed similar results for a related class of problems [4, 19]. These results indicate that an understanding of the interaction between knowledge and action (through knowledge-based algorithms) may facilitate the solution and understanding of important problems in distributed systems.

The impossibility of achieving common knowledge in many distributed systems appears to limit its utility when developing algorithms for these systems. In spite of this, Neiger and Toueg showed that solutions to certain problems may be developed for systems in which common knowledge *can* be achieved and that these solutions can be used correctly in systems in which it cannot [20]. They did so by "interpreting" knowledge in a nonstandard manner that did not alter the correctness of the algorithms under consideration. In running these algorithms, processors can

* Partial support for this work was provided by the National Science Foundation under Grants DCR-8601864, CCR-8909663, CCR-9106627, and CCR-9301454. This is a revised and expanded version of a paper that appeared in the *Proceedings of the Second Conference on Theoretical Aspects of Reasoning about Knowledge*, March 1988, pp. 295–308.

detect no inconsistency between this nonstandard interpretation and a standard one.

This technique is related to what Halpern and Moses termed *internal knowledge consistency*. This paper more thoroughly explores the notion of knowledge consistency and exhibits two types of such consistency. The first is that of Halpern and Moses and the second is a new form, called *uniform knowledge consistency*. It is shown that the latter is particularly useful when reasoning about the correctness of a program being run with a nonstandard knowledge interpretation. Several applications of knowledge consistency are then given, demonstrating its usefulness as a tool in the development of knowledge-based algorithms.

2. DEFINITIONS, ASSUMPTIONS, AND NOTATION

A distributed system is defined to be a set $\mathscr{P}$ of processors operating in an *environment* through which they communicate. Typically, the environment will be a message-passing medium, a shared memory, or some combination. At any given time, a processor has some *local state*, as does the environment. The state of the environment contains all information about the system's configuration that is not directly visible to the processors, including the contents of any communication channels, the shared memory, etc. Let $\mathscr{L}$ be the set of local states of the processors. A processor's local state may or may not include the value of a local *clock*, whose relationship to real time may or may not be specified. In general, the processor's local state is the input to the protocol being run (see below). The state of a distributed system, or *system state*, consists of the local states of the individual processors and that of the environment. Let $\mathscr{S}$ be the set of system states. If the system is in state s, then denote processor p's local state by $s(p)$.

A system's state changes over time due to *events* that occur in the course of an execution. Some events are the results of *commands* performed by the processors. Others are not explicitly controlled by the processors but are performed by the environment; these include the incrementing of local clocks and the delivery of messages by the environment. Let $\mathscr{C}$ be the set of commands executable by processors.

A specific execution of a system is described by a *history*. A history specifies the system states throughout the execution as well as the commands executed by the processors. It consists of two functions that describe the execution with respect to real time (assume that real times are elements of N, the set of natural numbers). The first is a *state history function* S, which describes the states through which the system passes; $\mathrm{S}\colon \mathbf{N} \mapsto \mathscr{S}$. If $\mathrm{S}(t) = s$, then the system was in state s at time t; in this case, $\mathrm{S}(t, p)$ will refer to $s(p)$. The second is a *command history function* C, which specifies the commands performed by processors; $\mathrm{C}\colon \mathbf{N} \times \mathscr{P} \mapsto \mathscr{C}$. This function is partial; $\mathrm{C}(t, p)$ may be undefined ($\perp$). If $\mathrm{C}(t, p) = c \in \mathscr{C}$, then processor p began executing command c at time t (when the system was in state $\mathrm{S}(t)$). If $\mathrm{C}(t, p) = \perp$, then, at time t, processor p is still busy executing an earlier invoked command.

Note that a history does not specify all *events* executed, only the commands performed by the processors. Other events, executed by the environment, are reflected by the changes in the system state. For example, if $\mathrm{S}(t)$ shows p's clock reading time 5 and $\mathrm{S}(t+1)$ shows p's clock reading time 6, we can infer that the environment updated the clock at time t. The type of events the environment executes depends on system being executed. For example, the environment in a system with perfectly synchronized clocks will never execute an event that updates the clock of one processor but not that of another.

For any history $\mathrm{H} = \langle \mathrm{S}, \mathrm{C} \rangle$, the states through which the system passes (given by S) must be compatible with the commands performed by the processors (given by C); for example, if $\mathrm{C}(t, p) =$ "send message m to q," then $\mathrm{S}(t)$ should reflect that the message has been sent. A formal definition of this compatibility depends on the particular system being considered and is thus beyond the scope of this paper.

Given a history $\mathrm{H} = \langle \mathrm{S}, \mathrm{C} \rangle$, define a function L that, given a processor and a real time, provides that processor's local *state-command history* through that time. This is the sequence of local states upon which the processor acts, each state paired with the associated command:

$$\mathrm{L}(p, t) = \begin{cases} \lambda & \text{if } t = 0 \text{ and } \mathrm{C}(0, p) = \perp; \\ \langle \mathrm{S}(0, p), c \rangle, & \text{if } t = 0 \text{ and } \mathrm{C}(0, p) = c \in \mathscr{C}; \\ \mathrm{L}(p, t-1), & \text{if } t > 0 \text{ and } \mathrm{C}(t, p) = \perp; \\ \mathrm{L}(p, t-1) \cdot \langle \mathrm{S}(t, p), c \rangle, & \text{if } t > 0 \text{ and } \mathrm{C}(t, p) = c \in \mathscr{C}. \end{cases}$$

Thus, $\mathrm{L}(p, 0)$ is the empty sequence λ if p executes no command at time 0; otherwise, it is the sequence containing that command with the local state from which it was executed. At later times, these sequences are extended based on the commands executed by the processor. Define p's state-command history for the entire run by setting $\mathrm{L}(p) = \lim_{t \to \infty} \mathrm{L}(p, t)$. Note that $\mathrm{L}(p)$ describes all of p's execution except for times at which p is busy; in a sense, it conveys the part of p's execution that is independent of the real timings of commands.

Two histories H_1 and H_2 are *state-command-equivalent* (written $\mathrm{H}_1 \cong \mathrm{H}_2$) if, for each processor $p \in \mathscr{P}$, $\mathrm{L}_1(p) = \mathrm{L}_2(p)$ (L_1 and L_2 are the state-command history functions for H_1 and H_2, respectively). If $\mathrm{H}_1 \cong \mathrm{H}_2$, then no processor can distinguish H_1 from H_2. As will be seen below, the notion of state-command-equivalent histories will be central to the definition of uniform knowledge consistency.

As indicated above, a particular distributed system is determined by the allowable behavior of the environment. Any history in which the environment behaves appropriately is considered an execution of that system. For this reason, a system is identified with the set of all histories that correspond to all such executions. Capital letters are used to denote such sets. In general, A will refer to the actual system being run, and I will refer to a system used to establish a knowledge interpretation.

A *protocol* Π is the distributed algorithm run by the processors. Given the local state of a processor, Π specifies the next command to be executed by that processor; $\Pi: \mathscr{P} \times \mathscr{L} \mapsto \mathscr{C}$. Thus, if processor p is in local state s when it performs a command, then it executes the command $\Pi(p, s)$.

History H is an *execution of protocol* Π if the commands executed by the processors in H are exactly those specified by Π, given the local states of the processors. That is,

$$\forall t \in \mathbf{N}\ \forall p \in \mathscr{P}[\mathrm{c}(t,p) \neq \perp \Rightarrow \mathrm{c}(t,p) = \Pi(p, \mathrm{s}(t,p))].$$

If S is a set of histories, define $S[\Pi]$ to be the subset of S containing only executions of Π.

Because the protocols defined here base their commands on the processors' local states, they cannot distinguish state-command-equivalent histories.

THEOREM 1. *If* $\mathrm{H}_1 \cong \mathrm{H}_2$ *and* H_1 *is an execution of* Π, *then so is* H_2.

Proof. Suppose that $\mathrm{H}_1 \cong \mathrm{H}_2$ and that H_1 is an execution of Π. The proof must show that H_2 is also an execution of Π.

Suppose that for some t and p, $\mathrm{c}_2(t,p) = c \in \mathscr{C}$. Let $s = \mathrm{s}_2(t,p)$. It suffices to show that $c = \Pi(p, s)$. By the definition of L, the pair $\langle s, c\rangle$ appears in the sequence $\mathrm{L}_2(p)$. Since $\mathrm{H}_1 \cong \mathrm{H}_2$, $\langle s, c\rangle$ also appears in $\mathrm{L}_1(p)$. Thus, for some time t', $c = \mathrm{c}_1(t', p)$ and $s = \mathrm{s}_1(t', p)$. Since H_1 is an execution of Π, $c = \Pi(p, s)$ as desired. Thus, H_2 is an execution of Π. ▌

In distributed systems, a problem to be solved can be specified by a predicate on histories. This is the problem's *specification*. For example, the *serializability* problem in distributed databases is specified by a predicate that is satisfied exactly by those histories of the database in which transactions are serializable. Protocol Π solves a problem with specification Σ in system S if, whenever processors run Π in S, Σ is satisfied. Formally, Π *satisfies* Σ *in* S if every history $S[\Pi]$ satisfies Σ.

Many problems in distributed systems are not sensitive to the real timings of different events in an execution but depend, for example, more on the relative ordering of events at different processors. Examples of such problems include transaction serializability, deadlock prevention, and stable-state detection. Recall that the state-command history of a processor contains all information about a processor's execution except these real timings. The definition of problem specifications can thus be specialized to be a predicate on state-command histories; call such specifications *internal specifications*. Formally, specification Σ is internal if

$$\forall \mathrm{H}_1, \mathrm{H}_2[\mathrm{H}_1 \cong \mathrm{H}_2 \Rightarrow (\Sigma(\mathrm{H}_1) \Leftrightarrow \Sigma(\mathrm{H}_2))];$$

that is, any two state-command-equivalent histories either both satisfy or both fail to satisfy Σ.

Not all specifications are internal. Consider a specification Σ that requires all processors to begin executing their second command at real time 17. Let $\mathrm{H}_1 = \langle \mathrm{S}_1, \mathrm{C}_1\rangle$ be some history that satisfies Σ. Let $\mathrm{H}_2 = \langle \mathrm{S}_2, \mathrm{C}_2\rangle$ be a history defined as follows:

$$\mathrm{s}_2(t,p) = \begin{cases} \mathrm{s}_1(t,p) & \text{if } t < 17 \\ \mathrm{s}_1(t-1,p) & \text{if } t > 17 \end{cases} \quad \text{and}$$

$$\mathrm{c}_2(t,p) = \begin{cases} \mathrm{c}_1(t,p) & \text{if } t < 17 \\ \perp & \text{if } t = 17 \\ \mathrm{c}_1(t-1,p) & \text{if } t > 17. \end{cases}$$

It should be clear that $\mathrm{H}_1 \cong \mathrm{H}_2$ but, in H_2, all processors begin their second command only at time 18, so H_2 does not satisfy Σ.

Consider problems that can be solved in totally asynchronous systems (those in which there is no bound on relative processor speeds, message-passing delays, or on memory-access latencies). In such systems, there is no way to satisfy any requirements on the real-timings of events. This suggests that any specification satisfiable in such systems must be internal.

3. KNOWLEDGE IN DISTRIBUTED SYSTEMS

This section considers how knowledge can be ascribed to processors in distributed systems, different interpretations for such knowledge, and how such interpretations can be used in knowledge-based protocols.

3.1. *Definitions*

This section formally ascribes knowledge to processors in a distributed system and gives a language for a logical system that can express such knowledge.

Assume the existence of a language for expressing certain facts about the system without referring to the knowledge of processors. These are *ground facts* and express properties that are or are not true of specific points in executions of the system. A *point* is a pair $\langle \mathrm{H}, t\rangle$ that refers to history H at time t. With each ground fact φ is associated a set $\pi(\varphi)$ of points for which the fact is true. Ground facts are those such as "processor p sent message m to processor q," "the value of processor q's variable x is 5," and "processor r has halted." The truth of a ground fact should be independent of the system being run and depend only upon the point in the execution. The language of ground facts is then closed under the usual propositional connectives (e.g., $\wedge$ and $\neg$). A fact is *valid in a system* if it is true at all points in the system. It is *valid* if it is valid in all systems.

Halpern and Moses introduced the modality operators $\mathbf{K}_p$ (one for each $p \in \mathscr{P}$) to extend a language of ground facts [10]; $\mathbf{K}_p\varphi$ denotes that processor p "knows" fact φ. How these knowledge operators are interpreted is a major concern of this paper. In general, such an interpretation should have properties that facilitate the design of knowledge-based protocols (see Section 3.3 below). Ideally, knowledge operators satisfy the properties of the modal logic S5 [9]:

A1: the *knowledge axiom*: $\mathbf{K}_p\varphi \Rightarrow \varphi$;[1]

A2: the *consequence closure axiom*: $\mathbf{K}_p\varphi \wedge \mathbf{K}_p(\varphi \Rightarrow \psi) \Rightarrow \mathbf{K}_p\psi$;

A3: the *positive introspection axiom*: $\mathbf{K}_p\varphi \Rightarrow \mathbf{K}_p\mathbf{K}_p\varphi$;

A4: the *negative introspection axiom*: $\neg\mathbf{K}_p\varphi \Rightarrow \mathbf{K}_p\neg\mathbf{K}_p\varphi$; and

R1: the *rule of necessitation*: if φ is valid in the system, then $\mathbf{K}_p\varphi$ is valid in the system.[1]

Higher levels of knowledge are defined based on the operators $\mathbf{K}_p$. $\mathbf{E}_G\varphi$ (everyone in group G knows φ) is $\bigwedge_{p\in G}\mathbf{K}_p\varphi$. If $\mathbf{E}_G^1\varphi \equiv \mathbf{E}_G\varphi$ and $\mathbf{E}_G^{m+1}\varphi \equiv \mathbf{E}_G(\mathbf{E}_G^m\varphi)$, then $\mathbf{C}_G\varphi$ (φ is *common knowledge* to group G) is equivalent to $\bigwedge_{m\geq 1}\mathbf{E}_G^m\varphi$.[2]

Common knowledge and variants thereof have been shown to be necessary for the execution of coordinated actions in distributed systems [4, 6, 11, 18, 19, 21]. However, Halpern and Moses showed that, in systems in which there is uncertainty in the behavior of the processors (e.g., due to failures) or of the environment (e.g., due to asynchrony in the message-passing medium), common knowledge of many interesting facts may be impossible to achieve. Because of these impossibility results, it is important to understand when achieving true common knowledge is *not* necessary and when a suitable weakening may suffice. This paper addresses that issue by examining different knowledge interpretations and how and when they can be used consistently.

3.2. *Knowledge Interpretations*

This section considers different interpretations of the knowledge operators $\mathbf{K}_p$. A knowledge interpretation $\mathscr{I}$ is a function from points to truth valuations on formulas. $(\mathscr{I}, \text{H}, t) \models \varphi$ indicates that φ holds at point $\langle \text{H}, t\rangle$ under interpretation $\mathscr{I}$. This paper considers only interpretations $\mathscr{I}$ that satisfy the following:

(1) if φ is a ground fact then $(\mathscr{I}, \text{H}, t) \models \varphi$ if and only if $\langle \text{H}, t\rangle \in \pi(\varphi)$;

(2) $(\mathscr{I}, \text{H}, t) \models \varphi \wedge \psi$ if and only if $(\mathscr{I}, \text{H}, t) \models \varphi$ and $(\mathscr{I}, \text{H}, t) \models \psi$;

(3) $(\mathscr{I}, \text{H}, t) \models \neg\varphi$ if and only if $(\mathscr{I}, \text{H}, t) \not\models \varphi$; and

(4) if $\text{s}(t, p) = \text{s}'(t', p)$, then $(\mathscr{I}, \text{H}, t) \models \mathbf{K}_p\varphi$ if and only if $(\mathscr{I}, \text{H}', t') \models \mathbf{K}_p\varphi$.

(1) to (3) ensure that $\mathscr{I}$ is consistent with the meaning of ground facts, conjunction, and negation. (4) states that a processor's knowledge must always be a function of its local state. For that reason, a knowledge interpretation can be considered a function from local states to a "knowledge valuation" of formulas.

Many researchers have found it useful to develop knowledge interpretations that are defined based on a particular system. Processor knowledge is based on the facts a processor would know when running in a specified system. This paper refers to such interpretations as *system-based interpretations*, and the remainder of the paper concentrates on such interpretations. If I is a system, let $\mathscr{I}_I$ denote the system-based interpretation based on system I, which need not equal A, the system being run. The truth valuation of formulas based on $\mathscr{I}_I$ is defined inductively on the structure of formulas: ground facts, conjunctions, and negations are given by (1) to (3) above. The operators $\mathbf{K}_p$ are interpreted as follows:

(4′) $(\mathscr{I}_I, \text{H}, t) \models \mathbf{K}_p\varphi$ if and only if $\forall \langle \text{H}', t'\rangle \in I \times \text{N}$ $[\text{s}(t, p) = \text{s}'(t', p) \Rightarrow (\mathscr{I}_I, \text{H}', t') \models \varphi]$

(note that (4′) is consistent with (4) above). This is the only part of the definition of $\mathscr{I}_I$ that explicitly depends on I. Because the operators $\mathbf{E}_G$ and $\mathbf{C}_G$ are defined in terms of the $\mathbf{K}_p$, $\mathscr{I}_I$ also interprets formulas using these operators.

For the most part, researchers have used an interpretation $\mathscr{I}_A$ based on the actual system A being run; in such cases, all of S5 holds. The following lemma generalizes this fact by considering a situation where the interpretation $\mathscr{I}_I$ is not necessarily based on A and indicates the cases in which different parts of S5 hold.

LEMMA 2. *Let A and I be two systems. Consider knowledge interpretation $\mathscr{I}_I$ when used in system A. Then*

1. *the operators $\mathbf{K}_p$ satisfy axioms* A2, A3, *and* A4 *of* S5;
2. *if $I \subseteq A$, then rule* R1 *holds*;
3. *if $A \subseteq I$, then axiom* A1 *holds*.

Proof. Suppose that $(\mathscr{I}_I, \text{H}, t) \models \mathbf{K}_p\varphi \wedge \mathbf{K}_p(\varphi \Rightarrow \psi)$. Then φ and $\varphi \Rightarrow \psi$ hold at all points in $I \times \text{N}$ in which p has

[1] $\varphi \Rightarrow \psi$ is shorthand for $\neg(\varphi \wedge \neg\psi)$.

[2] Given that the knowledge operators require an interpretation to be defined, R1 is more appropriately stated "if φ is valid in the system under a given interpretation, then $\mathbf{K}_p\varphi$ is valid in the system under that same interpretation."

[3] Halpern and Moses defined $\mathbf{C}_G\varphi$ to be the greatest solution to the fixed point equation $X \equiv \mathbf{E}_G(\varphi \wedge X)$ and showed that this definition is equivalent to the infinite conjunction given above.

the same local state as at $\langle \text{H}, t \rangle$. By the definition of $\Rightarrow$ (see footnote 1), ψ must also hold at all such points, so $(\mathscr{I}_I, \text{H}, t) \models \mathbf{K}_p \psi$, and A2 holds. A3 and A4 hold because equality of p's local states is an equivalence relation over $I \times \mathbf{N}$, regardless of A.

If $I \subseteq A$, then R1 holds because, if φ is valid in A (under $\mathscr{I}_I$), then it must also be valid in I (under $\mathscr{I}_I$). Standard reasoning can now be used to show that $\mathbf{K}_p \varphi$ is valid in A. If $A \subseteq I$, then A1, the knowledge axiom, also holds: if $\langle \text{H}, t \rangle \in A \times \mathbf{N}$ and $(\mathscr{I}_I, \text{H}, t) \models \mathbf{K}_p \varphi$, then $(\mathscr{I}_I, \text{H}', t') \models \varphi$ for all points $\langle \text{H}', t' \rangle \in I \times \mathbf{N}$ such that $\text{s}(t, p) = \text{s}'(t', p)$. Since $A \subseteq I$, $\langle \text{H}, t \rangle$ is one of these points and thus, $(\mathscr{I}_I, \text{H}, t) \models \varphi$. ▌

Lemma 2 has the following corollary.

COROLLARY 3. *Consider knowledge interpretation $\mathscr{I}_I$ when used in system I. Then the operators $\mathbf{K}_p$ satisfy all of* S5.

Axiom A1 need not be satisfied if I is a proper subset of A. For example, let A be an arbitrary distributed system, with no restrictions on the accuracy or synchronization of local clocks. Let I be a similar system in which clocks are always perfectly synchronized such that $I \subseteq A$. Suppose that system A is being run and let $\langle \text{H}, t \rangle \in A \times \mathbf{N}$ be a point at which processor p's clock shows 5. If $\varphi =$ "processor q's clock shows 5" then $(\mathscr{I}_I, \text{H}, t) \models \mathbf{K}_p \varphi$. This is because system I, in which clocks are perfectly synchronized, is being used to interpret the operator $\mathbf{K}_p$. Since H itself need not be in I, it may be that q's clock does *not* show 5 at $\langle \text{H}, t \rangle$, even though p "knows" that it does.

3.3. *Knowledge-Based Protocols*

Halpern and Fagin introduced *knowledge-based protocols* for distributed systems [8]. This section considers an adaptation of their work.

Section 2 defined protocols to map local states to commands. A knowledge-based protocol can use a processor's knowledge (as well as its state) to determine the next event to be executed. Such protocols are written in an extended language that allows the use of the knowledge operators $\mathbf{K}_p$, $\mathbf{E}_G$, and $\mathbf{C}_G$ to specify tests on the system state.

Recall that, by property (4) of knowledge interpretations, the facts that a processor knows are dependent only upon its state and the chosen interpretation. Thus, a knowledge-based protocol Π can be defined as a function that, given a processor's state and a knowledge interpretation, determines the next event to be executed by that processor.

History H is an *$\mathscr{I}$-execution of knowledge-based protocol Π* if processors execute exactly the events specified by protocol Π, using knowledge interpretation $\mathscr{I}$. That is,

$$\forall t \in \mathbf{N}\ \forall p \in \mathscr{P} [\text{C}(t, p) \neq \bot \Rightarrow \text{C}(t, p) = \Pi(p, \text{s}(t, p), \mathscr{I})].$$

If S is a set of histories, define $S[\Pi, \mathscr{I}]$ to be those histories in S that are $\mathscr{I}$-executions of Π. If all histories in $S[\Pi, \mathscr{I}]$ satisfy specification Σ, then *Π $\mathscr{I}$-satisfies Σ in S*.

An analogue to Theorem 1 holds for knowledge-based protocols.

THEOREM 4. *If $\text{H}_1 \cong \text{H}_2$ and H_1 is an $\mathscr{I}$-execution of Π, then so is H_2.*

Proof. Similar to the proof of Theorem 1. ▌

As stated earlier, it seems most natural to execute a knowledge-based protocol using a knowledge interpretation based upon the system being run; this interpretation will satisfy S5. There are occasions, however, when it may be useful to base knowledge upon other systems. These are discussed in Section 4.

In most cases, the knowledge-based protocol being run is itself common knowledge to the processors. That is, each processor will know what protocol the others are running. To capture this knowledge, it is appropriate to use an interpretation that is based not on the set of all runs of the system but on the set of runs of the system *that are executions of Π*. Although there is a circularity in the resulting definitions (the set of executions depends on the chosen interpretation, which in turn depends on the set of executions), Halpern and Fagin have shown that this set is well-defined if processor clocks are perfectly synchronized and processors' commands depend only upon knowledge of the past.

4. KNOWLEDGE CONSISTENCY

Halpern and Moses [10] defined a notion of "internal knowledge consistency." This section formalizes this and other forms of knowledge consistency.

4.1. *Two Definitions*

Knowledge consistency is a property that a knowledge interpretation has with respect to a particular system. Informally, an interpretation is consistent with respect to a given system if it is "appropriate" to use that interpretation when running a protocol in the system. A knowledge interpretation is consistent for a system if the processors running a knowledge-based protocol in that system can never detect any inconsistency based on that interpretation.

Consider the use of system-based interpretation $\mathscr{I}_I$, as defined above, when running in a system A. By Lemma 2, $\mathscr{I}_I$ must satisfy all of S5 with the possible exceptions of A1 (the knowledge axiom) and R1 (the rule of necessitation). A processor can detect an inconsistency only if it can determine that the knowledge axiom does not hold. It can do this only by knowing a fact and its negation.

When considering system-based interpretations, knowledge consistency represents a relation between two systems. One of these, A, is the system actually being run. The other,

I, is the "ideal" system, used to interpret the knowledge operators. As defined by Halpern and Moses, knowledge interpretation $\mathcal{I}_I$ is *internally knowledge-consistent with system A* if the following holds:

$$\forall\langle \mathrm{H}, t\rangle \in (A \times \mathbf{N})\, \forall p \in \mathcal{P}\, \exists\langle \mathrm{H}_p, t_p\rangle \in (I \times \mathbf{N}) \\ [\mathrm{s}(t, p) = \mathrm{s}_p(t_p, p)]. \tag{1}$$

This states that, at any point in an execution of system A there is for each processor a point in system I at which that processor has the same local state. Because of this, no processor can ever detect that it is not running in the system I. Recall that the knowledge axiom holds when using $\mathcal{I}_I$ in system I. Thus, no processor running in A can ever detect that the knowledge axiom (using $\mathcal{I}_I$) does not hold. The following theorems formalize this by showing that this definition is necessary and sufficient to prevent processors from detecting any inconsistency.

THEOREM 5. *Suppose that $\mathcal{I}_I$ is internally knowledge-consistent with system A. Then, for any fact φ, it* cannot *be the case that, for some $\mathrm{H} \in A$, $t \in \mathbf{N}$, and $p \in \mathcal{P}$, $(\mathcal{I}_I, \mathrm{H}, t) \models \mathbf{K}_p\varphi \wedge \mathbf{K}_p\neg\varphi$.*

Proof. Suppose the contrary for some φ, H, t, and p. Since $\mathcal{I}_I$ is internally knowledge consistent with A, there is a point $\langle \mathrm{H}_p, t_p\rangle \in I \times \mathbf{N}$ such that $\mathrm{s}(t, p) = \mathrm{s}_p(t_p, p)$. Since $(\mathcal{I}_I, \mathrm{H}, t) \models \mathbf{K}_p\varphi \wedge \mathbf{K}_p\neg\varphi$, it must be that $(\mathcal{I}_I, \mathrm{H}_p, t_p) \models \varphi \wedge \neg\varphi$, which contradicts condition (3) of the definition of a knowledge interpretation. ∎

THEOREM 6. *Suppose that $\mathcal{I}_I$ is* not *internally knowledge-consistent with system A. Then, for some $\mathrm{H} \in A$, $t \in \mathbf{N}$, and $p \in \mathcal{P}$, $(\mathcal{I}_I, H, t) \models \mathbf{K}_p\varphi \wedge \mathbf{K}_p\neg\varphi$ for* all *facts φ.*

Proof. Since $\mathcal{I}_I$ is *not* internally knowledge-consistent with A, it must be that

$$\exists\langle \mathrm{H}, t\rangle \in (A \times \mathbf{N})\, \exists p \in \mathcal{P}\, \neg\exists\langle \mathrm{H}_p, t_p\rangle \in (I \times \mathbf{N}) \\ [\mathrm{s}(t, p) = \mathrm{s}_p(t_p, p)].$$

Let φ any fact. By the definition of $\mathcal{I}_I$, it is vacuously true that $(\mathcal{I}_I, \mathrm{H}, t) \models \mathbf{K}_p\varphi \wedge \mathbf{K}_p\neg\varphi$, as desired. ∎

Although internally knowledge-consistent interpretations are necessary and sufficient to prevent inconsistencies from being detected, they do not provide some important properties that are useful in the design of knowledge-based protocols. For example, note that, for any point $\langle \mathrm{H}, t\rangle \in A \times \mathbf{N}$, each processor $p \in \mathcal{P}$ may have a *different* point $\langle \mathrm{H}_p, t_p\rangle$ in $I \times \mathbf{N}$ at which it has the same local state; there is no guarantee that $\mathrm{H}_p = \mathrm{H}_q$ for distinct p and q. Furthermore, if the definition gives $\langle \mathrm{H}_1, t'_1\rangle$ for $\langle \mathrm{H}, t_1\rangle$ (with $\mathrm{s}(t_1, p) = \mathrm{s}_1(t'_1, p)$) and $\langle \mathrm{H}_2, t'_2\rangle$ for $\langle \mathrm{H}, t_2\rangle$ (with $\mathrm{s}(t_2, p) = \mathrm{s}_2(t'_2, p)$), for two distinct times there is no guarantee that $\mathrm{H}_1 = \mathrm{H}_2$. The histories in I that p considers possible may change with time.[4] There are cases when one needs a stricter notion of knowledge consistency than this. For example, it may be necessary for all processors to believe that the same history of the ideal system is taking place. This motivates the idea of *uniform* knowledge consistency.

$\mathcal{I}_I$ is *uniformly knowledge-consistent with system A* if

$$\forall \mathrm{H} \in A\, \exists \mathrm{H}' \in I[\mathrm{H} \cong \mathrm{H}']. \tag{2}$$

Given any history H in the actual system A, there is (at least) one history of a system I that appears the same as H *to all processors at all times* and in which all processors act as they do in H. Furthermore, all processors execute the same commands (in the same order) in both histories; this is not a serious restriction, as a processor's local state often will encode all the commands it has performed. This definition is strictly stronger than the previous. Thus, the remarks regarding the knowledge axiom apply to it as well, as does Theorem 5 (of course, Theorem 6 does not apply).

(There is another property that makes uniform knowledge consistency more useful. As expressed, both forms of consistency are defined as relations between systems; they do not consider the protocols that processors are running. Suppose a processor is running knowledge-based protocol Π in system A, with a knowledge interpretation based on system I. In the case of internal knowledge consistency, it may be the that the history H_p, used to form point $\langle \mathrm{H}_p, t_p\rangle$ in Eq. (1), is not a history of Π. In this case, p could use its knowledge of Π to detect an inconsistency. This cannot occur with uniform knowledge consistency. By Theorem 4, the history H' used in Eq. (2), must be an execution of Π.)

Consider the following example from the domain of replicated databases. Let I be a system in which all transactions are performed serially and in the same order at each processor. Let A be a system that ensures *one-copy serializability* [5].[5] If a processor's view includes no more than the transactions that it has initiated and any results returned by them from the database, then $\mathcal{I}_I$ is uniformly knowledge-consistent with A.

4.2. *Using Knowledge Consistency*

The use of knowledge-consistent interpretations can simplify the design of knowledge-based protocols because an

[4] Some systems are such that processors never "forget" facts that they once knew. This is the case if processor p's local state at time t in history H, $\mathrm{s}(t, p)$, includes $\mathrm{L}(p, t-1)$, p's entire sequence of states and commands before point $\langle \mathrm{H}, t\rangle$. In that case, each processor has at least one history that it always believes to be possible.

[5] This essentially means that running a set of transactions in A has the transactions return the same results that they would if they had been run in the same serial order at all sites.

appropriately chosen knowledge interpretation can allow a protocol designer to make certain simplifying assumptions. That is, he or she can assume that processors can learn facts that they would (or could) not know when using a standard interpretation. Specifically, if system I is a subset of system A then, using $\mathcal{I}_I$ as a knowledge interpretation, a processor may act as if it knows facts that it would not if it used interpretation $\mathcal{I}_A$. Although this may appear to allow the designer to make spurious (and potentially dangerous) assumptions, knowledge consistency can be used to show that making such assumptions does not invalidate the correctness of the resulting protocol.

For example, let A be a system with asynchronous message passing (in which there is no bound on delivery times) and let I be a system in which messages are delivered exactly one second after they are sent and suppose that $\mathcal{I}_I$ is internally knowledge-consistent with A.[6] By using $\mathcal{I}_I$ processors can "know" facts that they would not if using $\mathcal{I}_A$: using $\mathcal{I}_I$, a processor knows that, if one second has elapsed since it sent a message, then the message has been delivered; this is not true when using $\mathcal{I}_A$. If $\mathcal{I}_I$ is truly knowledge-consistent with A, the processor can never detect that its message was not promptly delivered. Thus, any commands based on this "knowledge" should not be inconsistent.

Internal knowledge consistency guarantees that at no point in any history can any processor detect an inconsistency. This does not mean, however, that one can always use internally knowledge-consistent interpretations to simplify protocol design. One must also consider together all commands of all processors. The correctness of a knowledge-based protocol may depend upon facts that cannot be discerned by any single processor at a given time; specifically, it may depend upon the states through which the system passes in an execution of the protocol. However, when solving problems with internal specifications, a *uniformly* knowledge-consistent interpretation can be used to simplify the develop of a solution:

THEOREM 7. *Let I and A be two systems such that $\mathcal{I}_I$ is uniformly knowledge-consistent with A and let Σ be an internal specification. If knowledge-based protocol Π $\mathcal{I}_I$-satisfies Σ in I, then Π also $\mathcal{I}_I$-satisfies Σ in A.*

Proof. Let $\mathrm{H} \in A$ be an $\mathcal{I}_I$-execution of Π. Since $\mathcal{I}_I$ is uniformly knowledge-consistent with A, there is an $\mathrm{H}' \in I$ such that $\mathrm{H}' \cong \mathrm{H}$. By Theorem 4, H' is also an $\mathcal{I}_I$-execution of Π. Since Π $\mathcal{I}_I$-satisfies Σ in I, history H' satisfies Σ. Σ is internal so history H also satisfies Σ. Since H was chosen arbitrarily, Π must $\mathcal{I}_I$-satisfy Σ in A. ▌

Theorem 7 indicates that uniformly knowledge-consistent interpretations can be used when developing solutions to problems with internal specifications. In fact, the protocol designer need only prove the protocol to be correct in the ideal system; Theorem 7 guarantees its correctness in the actual system. The abstraction of an ideal system provides a designer with additional properties that simplify the design task. Thus, the use of uniform knowledge consistency can simplify the derivation of solutions to problems with internal specifications.

Consider again the example of replicated databases from Section 4.1, where I is a system in which transactions are truly serial, and A, where transactions are one-copy serializable. In general, the correctness of a replicated database depends only on the views of the database that processors observe in response to the transactions that they execute; it is not hard to see that this can be specified internally. Thus, a knowledge-based protocol, written to process transactions in system I, will run correctly in system A if $\mathcal{I}_I$ is used as the knowledge interpretation. Since it is much simpler to design such a protocol if transactions are executed serially, this simplifies the design of such systems. Knowledge-based protocols can be designed for these systems and proven correct with the simplifying assumption that transactions are executed serially.

5. APPLICATIONS OF KNOWLEDGE CONSISTENCY

This section considers some cases in which knowledge consistency can be applied in distributed systems.

5.1. *Granularity of Perception*

Recall that each processor perceives only the part of the system state that is its local state. Because of this, the granularity at which a processor perceives changes in the system state may be different from the granularity at which these changes actually occur. For example, one command, as executed by a processor, may be implemented by several distinct events (determined by the environment), each one of which changes the system state and possibly the processor's local state. However, because the processor is "busy" during these implementing events, it cannot perceive (or at least act upon) these changes to its local state. In such cases, one may want to use an interpretation based upon a system in which each command is one *atomic* event.

Fischer and Immerman consider two related systems that allow processors different views of the system state [7]. In one of these, C (*coarse*), one processor command consists of sending and receiving messages to and from all other processors in the system, as well as changing the processor's local state. Processors execute these once per "tick" on a global clock. The second system, F (*fine*), considers a finer granularity of execution. The sending and receipt of each message is a separate event, as is the final change of local state. (A processor's state may also change after the

[6] The truth of this assumption depends on the specific systems A and I; it is made here for the benefit of this example.

individual sending and receiving events.) No assumptions are made regarding the synchronization of processors nor message transmission rates.[7] Nevertheless, execution proceeds in *rounds*: in each round, a processor first sends messages, then receives messages, and then changes its state. Note that, in both systems, a protocol consults a processor's current state and the messages it has just received in determining its new state and messages to send in the next round. Thus, the part of the system state perceived by a processor is the same in both cases: the current state and messages received in the current "round" (the intermediate local states in system F are not used by the protocols).

Fischer and Immerman observe that, in C, it is relatively easy to obtain common knowledge. This is because of guarantees provided by the system: for example, when executing a command, a processor *knows* that the messages it sent in the previous round have been received. Because F is a more loosely coupled system, it provides no such guarantees and, for this reason, common knowledge cannot be attained in F.

Fischer and Immerman argue that these two systems are, in a sense, isomorphic and that it is therefore counterintuitive that common knowledge can be achieved in one and not in the other. This apparent contradiction can be explained by considering knowledge consistent interpretations. Because of the nature of the protocols considered and the fact that no messages are ever lost, interpretation $\mathcal{I}_C$ must be uniformly knowledge-consistent with system F. Therefore, one may write knowledge-based protocols for C and then run them in F, evaluating knowledge as if in C.[8] Such protocols thus operate as if common knowledge can indeed be achieved and are still correct when used to solve problems with internal specifications. If some protocol Π $\mathcal{I}_C$-solves a *non-internal* specification Σ in C, then it might not do so in F. For example, Σ may specify that all processors send each round's messages simultaneously. This can be accomplished in C but not in F.

5.2. *Simulation of Restricted Systems*

The system upon which a knowledge interpretation is based is often not simply an "isomorphic" version of the system being run. Sometimes, it is a system with different (and usually more useful) properties. One may want to simulate the execution of such an "ideal" system while actually running a more practical one. For example, it may not be practical to execute transactions *serially* in a distributed database system; a system with one-copy serializability is practical and simulates one with serial executions.

Neiger and Toueg considered such cases in systems with different message-passing and clock properties [20]. R_a was defined to be such a system with asynchronous message passing and perfectly synchronized real-time clocks; L_a was a similar system with a form of *logical clocks* [14]. Consider an execution $\text{H} \in L_a$; define $RT(\text{H})$ to be an execution that is identical in that all processors execute the same commands from the same local states (and thus at the same *local* times) but in which processors' clocks always show real time. It is not hard to see that $\text{H} \cong RT(\text{H})$ and that $RT(\text{H}) \in R_a$. This means that $\mathcal{I}_{R_a}$ is uniformly knowledge-consistent with L_a and can be used when solving internal specifications in L_a.

Following this, they considered systems with synchronous message passing. They defined R_s to be such a system with perfectly synchronized clocks and L_s to be one in which clocks are only approximately synchronized but in which communication is modified so that clocks have the causality properties of logical clocks. Once again, $\mathcal{I}_{R_s}$ is uniformly knowledge-consistent with L_s and can thus be used when solving internal specifications in L_s.[9]

In the same paper, they defined a message passing primitive, called the *publication*, that, if implemented, would achieve common knowledge in the ideal systems R_a and R_s.[10] They implemented publications in L_a and L_s and showed that these implementations can be used as if they achieve true common knowledge. These arguments can be formalized by appealing to the uniform knowledge consistency of interpretations $\mathcal{I}_{R_a}$ and $\mathcal{I}_{R_s}$ with systems L_a and L_s, respectively.

5.3. *Distributed Shared Memory*

Another area in which knowledge consistency can play a role is the study of distributed shared memories. Here, researchers study and compare different abstractions of shared memory that may be implemented in distributed systems that lack a true shared memory. This section shows how these abstractions can be expressed using the formalisms developed here and how they can be understood in terms of knowledge consistency.

[7] The two systems described here, C and F, correspond to the "protocols" $\mathcal{B}$ and $\mathcal{A}$ of Fischer and Immerman. They do not define protocols as done in this paper but instead posit the existence of a function that determines the next (local) state and messages to be sent. Although they call F "asynchronous," some synchronization is provided by the fact that a processor can detect whether or not the messages it has sent have been received.

[8] Fischer and Immerman do approximately the same thing by introducing the operators $\mathbf{K}_p^S$, where S is the system upon which the interpretation of these operators is based.

[9] Although both system R_s and L_s have synchronous message passing, the bounds on message delays must be greater in R_s for the indicated knowledge consistency to hold.

[10] As defined, publications achieve *timestamped common knowledge* [10, 20] in any system; in systems with perfectly synchronized clocks, this is identical to true common knowledge.

In a shared-memory system, processors do not communicate by message passing but by reading and writing shared memory locations. Assume that the memory consists of some finite set $\mathscr{M}$ of locations. To access the memory, processors may execute two kinds of commands: processor p uses $Read_p(x)$ to read location x and $Write_p(x)\,v$ to write value v to x. The environment "responds" to such commands with two kinds of events: $ReadReturn_p(x)\,v$ reports to p that v is stored in x, while $WriteReturn_p(x)\,v$ signals to p that its write of v to x has completed. Histories of a shared-memory system have the following properties: after executing a memory-access command and before receiving a response, a processor initiates no commands. Furthermore, the environment returns only responses that "match" commands already executed by the processors. The remainder of this section considers *asynchronous* shared-memory systems. In these, there is no bound on relative processor speeds or on memory-access latencies (the latter being the delay between the initiation of a memory-access command and its response). Furthermore, processors do not have access to clocks that measure real time.

The consistency of a shared-memory system is determined by how its histories correspond to *linearizations* of their memory-access commands. A linearization of a history is a total order of its commands such that, if $ReadReturn_p(x)\,v$ is a response to some command $c = Read_p(x)$, then v is the value stored by the latest *Write* to x in the ordering that appears before c.

Ideally, the memory should appear as if processors are atomically accessing a single shared memory. Such a memory is *atomic* [16] or *linearizable* [13]. Let H be a history of a shared-memory system. Let c_1 and c_2 be two memory-access commands executed in H. If the environment's response to c_1 occurs before c_2 is initiated, we write $c_1 \rightarrow c_2$. This is a partial order that indicates that c_1 strictly precedes c_2 in real time. History H is atomic (or linearizable) if there is a linearization of H such that, if $c_1 \rightarrow c_2$, c_1 precedes c_2 in the linearization. A system provides atomic memory if all its histories are atomic.

While the abstraction of atomic memory is the closest to that of a real shared memory, it is quite expensive to implement. Attiya and Welch [3] have shown that *sequential consistency*, a related but weaker form of memory, can be implemented more efficiently. Sequential consistency was defined by Lamport [15] as follows. History H is sequentially consistent if there is a linearization of H such that, if c_1 and c_2 are memory-access commands by the same processor and c_1 is initiated first, c_1 precedes c_2 in the linearization. A system provides sequential consistency if all its histories are sequentially consistent.

An atomic memory system is an ideal abstraction for distributed memory systems; often, it is more practical to provide only sequential consistency. It turns out that a knowledge interpretation based on a system with atomic memory is uniformly knowledge consistent with a system with sequentially shared memory:

THEOREM 8. *Let A and I be systems with the same processors and memory locations such that A is sequentially consistent and I is atomic. Then $\mathscr{I}_I$ is uniformly knowledge-consistent with A.*

Proof. The proof must show that $\forall \text{H} \in A\ \exists \text{H}' \in I\ [\text{H} \cong \text{H}']$. Let H be a history in A. Since H is sequentially consistent, there is a linearization of H such that, if c_1 and c_2 are memory-access commands by the same processor and c_1 is initiated first, c_1 precedes c_2 in the linearization. Now define H′ as follows: if c is ith in the linearization, then it is initiated at time $2i-1$ and receives a response at time $2i$. The two histories differ only with respect to the real times at which events occur. History H′ is atomic as the same linearization exists and, by definition, respects the real-time ordering of the commands in H′. Thus, $\text{H}' \in I$. In both histories, each processor executes the same commands in the same order and receives the same responses. Thus, $\text{H} \cong \text{H}'$, as desired. ▌

Theorems 7 and 8 imply now that knowledge-based protocols for shared-memory systems can be developed with the assumption that memory is atomic and then run in systems with sequential consistency. If the knowledge interpretation used is based on atomic memory, then such protocols will correctly solve internal specifications.

Although sequential consistency can be implemented more efficiently than atomic memory, its implementations can still be quite costly [17]. For this reason, researchers have proposed a variety of even weaker forms of distributed shared memory. One example is *causal memory* [1]. This memory is based on a partial causal order of memory-access commands; this is similar to the causal order Lamport defined for message-passing systems [14]. A formal definition of causal memory is beyond the scope of this paper, but it differs from sequential consistency in the following ways:

- instead of a single linearization of all memory-access commands, there is a (possibly different) linearization for each processor;
- a processor's linearization does not include the *Read* commands of other processors; and
- each linearization must be consistent with the causal ordering of commands.

Causal memory can be implemented more efficiently than sequential consistency because its implementations require no blocking.

Unfortunately, a result such as Theorem 8 cannot be shown for causal memory. In some programs, processors may be able to detect that they are not running with sequential consistency. Thus, even internal knowledge consistency does not hold. However, there is a large class of programs that do not allow any inconsistencies to be detected. These are the *data-race free protocols*; Ahamad *et al.* provide a definition and the following result [2, Theorem 5].

THEOREM 9 [Ahamad *et al.*]. *If A is a system with causal memory, I is a system with sequential consistency, and Π is a data-race free protocol, $A[\Pi] \subseteq I[\Pi]$.*

The following is a corollary to Theorem 9.

COROLLARY 10. *Let A and I be systems with the same processors with the same processors and memory locations such that A provides causal memory and I is sequentially consistent. Let Π be a data-race free knowledge-based protocol. Then $\mathcal{I}_{I[\Pi]}$ is uniformly knowledge-consistent with $A[\Pi]$.*

Corollary 10 and Theorem 7 imply that data-race free knowledge-based protocols for shared-memory systems can be developed with the assumption that memory is sequentially consistent and then run in systems with causal memory. If the knowledge interpretation used is based on sequential consistency, then such protocols will correctly solve internal specifications. (In fact, the protocols will correctly solve any specification, because of the subset relation given in Theorem 9.)

6. CONCLUSIONS

The study of processor knowledge in distributed systems has been useful in the analysis of distributed algorithms. The development of knowledge-based protocols has permitted the design of algorithms that can take advantage of the knowledge requirements of different problems.

How a processor determines, or interprets, its knowledge profoundly affects the behavior of a knowledge-based protocol. An interpretation is knowledge-consistent with a system if processors using it in that system can detect no inconsistencies. This paper has formally developed the notion of knowledge consistent interpretations and has shown that uniformly consistent knowledge interpretations can be used to simplify the design of knowledge-based protocols; a designer can assume that the distributed system being used had certain useful properties (e.g., perfectly synchronized clocks) and then correctly run his or her protocol in a system without such properties.

It has been shown that in systems with perfectly synchronized clocks, *timestamped common knowledge* is identical to true common knowledge [10, 20]. If a knowledge interpretation based on such a system is uniformly knowledge consistent with some practical system, then achieving timestamped common knowledge in the practical system is as good as achieving true common knowledge when solving problems with internal specifications. This can be explained by appealing to uniform knowledge consistency. There are other weakenings of common knowledge that can be achieved in practical distributed systems; in addition to those described by Halpern and Moses [10], there is also *concurrent common knowledge*, introduced by Panangaden and Taylor [22]. New formulations of knowledge consistency may be related to the difference between true knowledge and its other weakened forms.

ACKNOWLEDGMENTS

I thank Micah Beck, Phillip W. Hutto, Murray Mazer, Amitabh Shah, Patrick Stephenson, and Sam Toueg for reading and commenting on earlier drafts of this paper. In addition, the anonymous referees provided many helpful comments.

Received July 1992; final manuscript received March 1, 1994

REFERENCES

1. Ahamad, M., Burns, J. E., Hutto, P. W., and Neiger, G. (1991), Causal memory, *in* "Proceedings of the Fifth International Workshop on Distributed Algorithms" (S. Toueg, P. G. Spirakis, and L. Kirousis, Eds.), Lectures Notes in Computer Science, Vol. 579, pp. 9–30, Springer-Verlag, A revised and expanded version exists [2].
2. Ahamad, M., Neiger, G., Burns, J. E., Kohli, P., and Hutto, P. W. (1995), Causal memory: Definitions, implementation, and programming, *Distribut. Comput.* **9**, in press.
3. Attiya, H., and Welch, J. L. (1994), Sequential consistency versus linearizability, *ACM Trans. Computer System* **12**(2), 91–122.
4. Bazzi, A., and Neiger, G. (1992), The complexity and impossibility of achieving fault-tolerant coordination, *in* "Proceedings of the Eleventh ACM Symposium on Principles of Distributed Computing," pp. 203–214, ACM PRESS, New York.
5. Bernstein, A., and Goodman, N. (1986), Serializability theory for replicated databases, *J. Comput. Systems Sci.* **31**(3), 355–374.
6. Dwork, C., and Moses, Y. (1990), Knowledge and common knowledge in a Byzantine environment: Crash failures, *Inform. and Comput.* **88**(2), 156–186.
7. Fischer, M. J., and Immerman, N. (1986), Foundations of knowledge for distributed systems, *in* "Proceedings of the First Conference on Theoretical Aspects of Reasoning about Knowledge" (J. Y. Halpern, Ed.), pp. 171–185, Morgan Kaufmann, San Mateo, CA.
8. Halpern, J. Y., and Fagin, R. (1989), Modelling knowledge and action in distributed systems, *Distribut. Comput.* **3**(4), 159–177.
9. Halpern, J. Y., and Moses, Y. (1985), A guide to the modal logic of knowledge and belief, *in* "Proceedings of the Ninth International Joint Conference on Artificial Intelligence," pp. 480–490, Morgan Kaufmann, San Mateo, CA.
10. Halpern, J. Y., and Moses, Y. (1990), Knowledge and common knowledge in a distributed environment, *J. Assoc. Comput. Mach.* **37**(3), 549–587.
11. Halpern, J. Y., Moses, Y., and Waarts, O. (1990), A characterization of eventual Byzantine agreement, *in* "Proceedings of the Ninth ACM Symposium on Principles of Distributed Computing," pp. 333–346, ACM Press, New York.

12. Halpern, J. Y., and Zuck, L. D. (1990), A little knowledge goes a long way: Knowledge-based derivations and correctness proofs for a family of protocols, *J. Assoc. Comput. Mach.* **39**(3), 449–478.
13. Herlihy, M. P., and Wing, J. M. (1990), Linearizability: A correctness condition for concurrent objects, *ACM Trans. Programming Languages Systems* **12**(3), 463–492.
14. Lamport, L. (1978), Time clocks, and the ordering of events in a distributed system, *Comm. ACM* **21**(7), 558–565.
15. Lamport, L. (1979), How to make a multiprocessor computer that correctly executes multiprocess programs, *IEEE Trans. Comput.* **C-28**(9), 690–691.
16. Lamport, L. (1986), On interprocess communication; part I: Basic formalism, *Distribut. Comput.* **1**(2), 77–85.
17. Lipton, R. J., and Sandberg, S. (1988), "PRAM: A Scalable Shared Memory," Technical Report 180-88, Department of Computer Science, Princeton University.
18. Moses, Y., and Tuttle, R. (1988), Programming simultaneous actions using common knowledge, *Algorithmica* **3**(1), 121–169.
19. Neiger, G., and Bazzi, R. (1992), Using knowledge to optimally achieve coordination in distributed systems, *in* "Proceedings of the Fourth Conference on Theoretical Aspects of Reasoning about Knowledge" (Y. Moses, Ed.), pp. 43–59, Morgan Kaufmann, San Mateo, CA.
20. Neiger, G., and Toueg, S. (1993), Simulating synchronized clocks and common knowledge in distributed systems, *J. Assoc. Comput. Mach.* **40**(2), 334–367.
21. Neiger, G., and Tuttle, M. R. (1993), Common knowledge and consistent simultaneous coordination, *Distribut. Comput.* **6**(3), 181–192.
22. Panangaden, P., and Taylor, K. (1992), Concurrent common knowledge: Defining agreement for asynchronous systems, *Distribut. Comput.* **6**(2), 73–94.